

LA-14094-T

Thesis

Approved for public release;
distribution is unlimited.

Distributed Sensor Network Software Development Testing through Simulation

This thesis was accepted by the Department of Computer Science, University of New Mexico, Albuquerque, New Mexico, in partial fulfillment of the requirements for the degree of Master of Science. The text and illustrations are the independent work of the author and only the front matter has been edited by the IM-1 Writing and Editing Staff to conform with Department of Energy and Los Alamos National Laboratory publication policies.

©2003, Sean M. Brennan

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the Regents of the University of California, the United States Government nor any agency thereof, nor any of their employees make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Regents of the University of California, the United States Government, or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Regents of the University of California, the United States Government, or any agency thereof. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

LA-14094-T

Issued: December 2003

Distributed Sensor Network Software
Development Testing through Simulation

Sean M. Brennan

**DISTRIBUTED SENSOR NETWORK SOFTWARE DEVELOPMENT
TESTING THROUGH SIMULATION**

by

SEAN M. BRENNAN

B.A., University of Iowa, 1994

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

**Master of Science
Computer Science**

The University of New Mexico
Albuquerque, New Mexico

November 2003

Sean M. Brennan

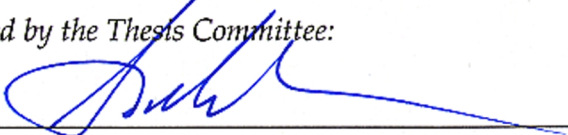
Candidate


Computer Science

Department

This thesis is approved, and it is acceptable in quality
and form for publication on microfilm:

Approved by the Thesis Committee:

 , Chairperson


David A. Bader

Accepted:

Dean, Graduate School

Date

Dedication

For Annely, Guendalyn, and Sedrik.

Acknowledgments

I would like to thank my colleagues and advisors for their valuable input, including - but not limited to - Barney Maccabe, Jared Dreicer, Bob Nemzek, and David Torney.

Funding for this research was provided by the ISR (formerly NIS) and STB-EPO divisions at Los Alamos National Laboratory.

This unclassified document is released as a Los Alamos National Laboratory LA-Series Report, number LA-14094-T, and can be found in multiple formats at <http://www.cs.unm.edu/~sbrennan/docs/>.

*Approved for public release;
distribution is unlimited.*

**DISTRIBUTED SENSOR NETWORK SOFTWARE DEVELOPMENT
TESTING THROUGH SIMULATION**

by

SEAN M. BRENNAN

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

**Master of Science
Computer Science**

The University of New Mexico
Albuquerque, New Mexico

November 2003

**DISTRIBUTED SENSOR NETWORK SOFTWARE DEVELOPMENT
TESTING THROUGH SIMULATION**

by

SEAN M. BRENNAN

B.A., University of Iowa, 1994

M.S., Computer Science, University of New Mexico, 2003

Abstract

The distributed sensor network (DSN) presents a novel and highly complex computing platform with difficulties and opportunities that are just beginning to be explored. The potential of sensor networks extends from monitoring for threat reduction, to conducting instant and remote inventories, to ecological surveys. Developing and testing for robust and scalable applications is currently practiced almost exclusively in hardware.

The Distributed Sensors Simulator (DSS) is an infrastructure that allows the user to debug and test software for DSNs independent of hardware constraints. The flexibility of DSS allows developers and researchers to investigate topological, phenomenological, networking, robustness and scaling issues, to explore arbitrary algorithms for distributed sensors, and to defeat those algorithms through simulated failure. The user specifies the topology, the environment, the application, and any number of arbitrary failures; DSS provides the virtual environmental embedding.

Contents

List of Figures	xiii
List of Tables	xv
Glossary	xvi
1 Ubiquitous Computing and Distributed Sensor Networks	1
1.1 The Third Age of Computing	2
2 Background	6
2.1 Applied DSNs	6
2.1.1 Habitat Data Collection on Great Duck Island	7
2.1.2 Structural Integrity Monitoring	7
2.1.3 All-seeing Argus	8
2.2 Development Guidelines	9
2.2.1 Topologies and Node Deployment	9

Contents

2.2.2	Computation	11
2.2.3	Data Routing	13
2.3	DSN Operating Systems	15
2.3.1	TinyOS	16
2.3.2	EYES OS	17
2.4	Fault-Tolerance and Dependability Testing	18
2.4.1	Failure Modes	18
2.4.2	Dependability Testing	20
3	Design Approach	24
3.1	DSS System Architecture	25
3.1.1	User Interface	26
3.1.2	SimCore	27
3.1.3	Wireless Channel	29
3.1.4	Algorithm Plug-in	29
4	Implementation	32
4.1	Realizing the System Architecture	33
4.1.1	The User Interface	34
4.1.2	The Event Priority Queue	37
4.1.3	The Scripted EnviroSim	37

Contents

4.1.4	Configuration with XML	38
4.1.5	FailSim	39
4.1.6	The Wireless Channel Model	39
4.2	The Application Programming Interface	40
4.3	Features	40
5	Validation Using Experimental Results	43
5.1	The Empirical Experiment	45
5.1.1	August 13	46
5.1.2	July 11, August 14, and August 28	49
5.2	The Virtual Experiment	50
6	Related Work	55
6.1	TOSSIM	55
6.2	SWAN	56
6.3	SensorSim	57
6.4	SensorSimII	58
7	Future Work on DSS	59
7.1	TinyOS Integration	59
7.2	Improved Wireless Channel Realism	60
7.3	Robotic Mobility	61

Contents

7.4	Environmental Geometry	61
7.5	Calibration Tools	62
7.6	Node Processor Emulation	62
8	Discussion	64
8.1	Behavioral Simulation and Visualization	64
8.2	Information Dissemination	65
8.3	Using DSS	67
	Appendices	70
1	TDOA Algorithm	70
2	Distributed Sensors Simulator HOWTO	73
1	Introduction	73
1.1	Acknowledgments	74
2	Preparation	75
2.1	DSS Setup	75
2.2	Considerations	75
3	Running DSS	76
3.1	... Locally	77
3.2	... Remotely	77

Contents

3.3	Cluster Computing	79
4	Configuring the simulation	79
4.1	Create the Application(s)	79
4.2	Compose the Phenomenology Script(s)	81
4.3	Specify the Node Topology	84
4.4	Specify the Source(s)	87
4.5	Specify any Failures	90
3	Example Phenomenology Script	93
	References	96

List of Figures

2.1	Simple Sensors in a Star Configuration	10
2.2	Smarter Sensors Slaved to a Compute Center	11
3.1	DSS Architecture: Overall	26
3.2	EnviroSim Event Queue Interaction	28
3.3	DSS Architecture: Remote Machine	30
4.1	Simulator Implementation Objects	33
4.2	DSS Basic Screen	36
5.1	TDOA Locating Four Shooters (A, B, C, D)	46
5.2	TDOA Using Sub-Optimal Topologies: July 11	47
5.3	TDOA Using Sub-Optimal Topologies: August 14	48
5.4	TDOA Using Sub-Optimal Topologies: August 28	49
5.5	Error Distances (August 13)	51
5.6	Error Distances	52

List of Figures

8.1	Radiation Detection	67
A-2.1	Simulation Topology Screen	78
A-2.2	Network Configuration Screen	84
A-2.3	Group Config Screen	85
A-2.4	Source Configuration Screen	88

List of Tables

5.1	TDOA Experimental Results: August 13	46
5.2	TDOA Simulation Results: August 13	51

Glossary

API	Application Programming Interface. A software code interface that allows programmers to utilize an external code source such as a library.
DSN	Distributed Sensor Network. Smart sensors coordinated over a geographical area to provide environmental information services. Data is commonly fused in situ or otherwise combined to provide a comprehensive picture for the user.
DSS	Distributed Sensors Simulator. A software development tool that aids in the creation of applications for DSNs.
LANL	Los Alamos National Laboratory. A federal laboaratory in northern New Mexico.
OS	Operating System. Software that abstracts away hardware complexities, schedules processes, and may allocate storage or other resources. Typically this is implemented as a single program or groups of programs that manage all other applications.
RF	Radio Frequency. Often refers to the wireless channel as a whole rather than a single frequency.

Glossary

SQL	Structured Query Language. A programming language typically used for database access.
TDOA	Time Difference of Arrival. A method of calculating an event's source location using wavefront or ray arrival times at diverse locations.
TDOA-CLS	TDOA with Constrained Least Squares. The TDOA method combined with an added refinement that filters out errors.
TinyOS	The most common operating system for DSNs. It defies common notions of what constitutes an OS, but not the definition of an OS.
WCM	Wireless channel model. A software model of the behavior of bit broadcasts over radio frequencies; a module of DSS.

Chapter 1

Ubiquitous Computing and Distributed Sensor Networks

In 1999, the Smart Dust initiative was launched from the work of UC Berkeley professor Kris Pister with major funding from the Defense Advanced Research Projects Agency (DARPA). The project's three-year goal was to create an autonomous processing and communication platform, of just a single cubic millimeter in size, for a wide variety of sensors. The enormous potential for defensive and offensive capabilities of such systems obviously drew the attention of DARPA. Yet if, as has been predicted, these devices become incredibly inexpensive, civilian applications of this technology are likely to become as widespread and revolutionary, and as divorced from the project's original intent, as that other DARPA project known as the Internet. This revolution has been dubbed ubiquitous computing.

With such an emphasis on small size, these devices - or at least those at the leading edge of development - are of necessity slow to process and communicate data. The processing and memory limitations of these devices are reminis-

cent of their large counterparts that were standard well over two decades ago. This forces a new paradigm in processing methods, software development, and even problem-solving approaches just to clear the high bar of what is expected of computing platforms today.

The next section details how the vision of ubiquitous computing affects DSN development and provides this development with guidelines for the future of human/computer interactions.

1.1 The Third Age of Computing

For over a decade, the Next Big Thing in computing technology has been something now dubbed ubiquitous computing. For many people the idea of a nearly omnipresent computer is as distantly futuristic as a manned mission to Jupiter.

The truth is that this "third paradigm"¹ has been a long time coming, and is closer to realization than might be apparent at first.

But just what is a ubiquitous computer? Mark Weiser, considered the Father of Ubiquitous Computing, calls it the transparent computer. "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it," [39]. Ubiquitous computing means that the whole environment will spew pre-digested information tailored to a user's task at hand and be even more aware of the surroundings than the user. This vision rejects the current paradigm of the desktop metaphor as just a simile and in particular the demanding focal point of a mon-

¹Weiser attributes this phrase to visionary and Smalltalk creator Alan Kay. It refers to the progression toward greater intimacy of human/computer interactions first from mainframe computers, then to personal computers, and now to ubiquitous computers.

itor screen for all interactions ("who would ever use a desk whose surface area is only 9 [inches] by 11 [inches]?" [39]). Ubiquitous computers have emerged from the single box and recede into the background, merging with human computing needs in a much more natural way. Rather than force humans to adapt to the computer, the ubiquitous computer adapts to the needs of each individual user.

All this may sound like fantasy, but it is not just the stillborn brainchild of one person. Around the world many real, functional projects which are presented in conferences focus on the ubiquitous paradigm.

As computers emerge from their cases and meld into the woodwork, they tend to shrink in size and become tetherless. Currently the ubiquitous computer is most distinctly embodied in the DSN. While there are other interpretations of the vision, DSNs are the most widespread and promising. Just what constitutes a DSN is sometimes not so clear.

DSNs can be defined by an explication on the name:

- **Distributed** Firstly, DSNs are distributed - most obviously geographically. Ubiquity promises presence everywhere, so DSNs are spread out, but not underfoot. This far-flung embedding usually requires individual nodes to be long-lasting, low-power, and inexpensive. Distributed also implies that the processing is not located in any one place, which can easily lead to the tolerance of a variety of faults.
- **Sensing** As a result of being distributed, DSNs are positioned to sample a surrounding environment over a relatively wide area. Human/computer interaction mediated by the surroundings (rooms, furniture, doors, etc.) is a core tenet of ubiquity. Aside from DSNs, robotics is the only field that requires real-world awareness to such a degree. Such contextual immersion not only allows the computer to better interact with humans, but, more

importantly, it vastly increases the quality and quantity of truly useful information an embedded computer can yield.

- **Networked** Sensor distribution without some sort of coordination is pointless. Networked computing sensors can fuse and transform real-world data to produce information that would be otherwise unattainable. Networking unifies singular sensing nodes into a coherent whole.

A single node tends to be rather simple. Traditional remote sensors simply send off a raw detection signal to be processed. The next step up, the smart sensor, has onboard firmware which automates removal of nonlinearities and electrical noise from the detection signal before transmitting it to a processing unit. These compact smart sensors are usually enabled by Micro-Electro-Mechanical Systems (MEMS) technology which allows reduced size and cost and can increase accuracy and performance.

Smart Dust takes this progression one step further. The individual Smart Dust node can find and communicate with neighboring sensor nodes to form a DSN, and even process (not just pre-process) its own data. Begun by Pister at UC Berkeley, the Smart Dust project aims for sensor nodes of just one cubic millimeter; already, fully functional prototypes have shrunk to five cubic millimeters. The project continues today in pursuit of its original goal.

Size will soon no longer dictate low functionality. As Moore's Law continues with no end in sight, the laptop of today is the handheld of tomorrow, and today's palmtop is tomorrow's Smart Dust.

How is software developed for computers that can barely be seen? How are networks that scale to hundreds of thousands of nodes developed? The Distributed Sensor Simulator is a software development tool designed specifically for DSNs. It handles issues of sensor network development not addressed else-

where.

This document discusses both DSS and the relatively fledgling field of DSNs that the simulator embodies and explores. Chapter 2 summarizes the extensive background of DSNs and issues prevalent in DSN development. This includes issues in node topology, computation models, data routing, and software fault-tolerance.

Chapter 3 presents the design architecture of DSS, while Chapter 4 expands on certain aspects of the software implementation. Chapter 5 then demonstrates the validity of the software.

Chapter 6 notes similar projects that had some influence on the design of DSS, and Chapter 7 discusses where this project is going next.

Finally, Chapter 8 concludes with a discussion of behavioral visualization, wide-area data dissemination, and how DSS is currently being put to use.

Chapter 2

Background

Before detailing the architecture and implementation of DSS, the context from which it arises must be explained. This chapter samples some existing DSN systems, examines some of the most important issues of developing a DSN, and finally investigates application dependability and fault recovery. DSS provides for simulated explorations of all of these aspects of DSNs.

2.1 Applied DSNs

Talking about ubiquity and technological paradigm shifts is somewhat unreal, with an air of the purely fantastic. What is the state of DSNs right now and how close are the promises of ubiquity? The following three working DSN systems exemplify different means of data collection, fusion, and dissemination as discussed in later sections. These projects not only demonstrate the here-and-now of this technology, but they also point out that DSNs, and by extension ubiquitous computing, are *not* about size, they are about interaction.

2.1.1 Habitat Data Collection on Great Duck Island

On a small island off the coast of Maine, the College of the Atlantic operates a seabird habitat sanctuary. Great Duck Island is a favored location for Storm Petrels to breed and nest in burrows. Although the CoA seeks to study the incubation behavior and nesting choices of these birds, it has been observed elsewhere that any disruption by human presence dramatically increases the mortality rate of the unhatched. Entire colonies will even abandon a site if they are repeatedly disturbed. The CoA, in cooperation with UC Berkeley, installed a DSN of 32 nodes that reported light, temperature, pressure, and humidity for over four months [29]. Sensors were placed near nesting burrows, as well as in uninhabited areas, in order to help biological researchers determine the optimal microclimate for breeding.

Using multihop routing, sensor data is funneled back to a collection station where it is recorded, averaged, and displayed on the Internet. The project can still be seen in declining operation at www.greatduckisland.net. Individual nodes have multiple sensing devices, each of which is sampled at a relatively slow rate. Power conservation is a high priority to minimize human intervention in maintaining the DSN.

2.1.2 Structural Integrity Monitoring

Currently the typical method for determining the structural health of a building, particularly after an earthquake, is through time-consuming, costly, and imprecise visual inspection. Large expensive seismic accelerometers exist for detecting problems, but they are difficult to install and hence are used sparsely. Steve Glaser of UC Berkeley's Civil Engineering Department is exploring the use of small untethered detectors clustered around numerous key

structural points to yield much more complete, and more fine-grained information [10]. This allows local damage to be identified rapidly, thus increasing safety.

In this system, the sensor nodes are interested in only two events: a) tremors as detected by onboard accelerometers, and b) queries for data. The large majority of the lifetime of one of these nodes will be spent sleeping. Upon a seismic disturbance, the accelerometer, which is always powered, wakes the node which verifies and stores the data. The network can be queried at a later time to collect this data and determine, offline, the severity of the shock wave overall and estimate localized damage.

2.1.3 All-seeing Argus

While the previous two examples depended on small nodes, the Argus project at Duke University would not appear to be a DSN at all on the basis of size. Using an array of 64 cameras in a ring, Argus is a prototype telepresence system [7]. The cameras are hidden behind a backdrop, defining a circular space large enough to fit several people. The system yields either stereo image pairs or a volumetric data set of objects in this space. What makes this a DSN is the fact that each camera is coupled with its own high-speed processor. These 64 processors act in concert as a Beowulf-class cluster. It has unlimited power supply, relatively enormous bandwidth, and top-of-the-line processing. Yet, the system requires that it be hidden 'in the walls', and it has a one-to-one correspondence of sensor-to-processing/communicating resource. Data is shared across the cluster, and the system constructs its output of an image pair or volume data to ship out across a network in real time. This behaves just like a DSN, a very high-powered DSN, and a pointer to the future should Moore's Law continue to hold.

The common thread through these examples is that the knowledge gleaned from these DSNs is greater than a mere summing of data. It allows the big picture to be seen, even literally.

2.2 Development Guidelines

This section describes typical DSN development choices and their effect on performance. For a particular DSN application, node topology, computation strategies, and data routing may have a strong impact on that DSN's functionality and robustness. Achieving big picture results is not necessarily a straightforward process, but DSS is flexible enough to enable any of the variety of approaches covered here through its simulation configuration.

2.2.1 Topologies and Node Deployment

Sensors are distributed in order to maximize their exposure to the phenomena they seek to record. *How* they are distributed can have a great effect on their functioning.

There are three generic types of DSNs: a) classical, which involves direct sensor contact with a central processing station, b) multi-hop routing to a central processor, and c) DSNs with in situ processing and no central station.

In a classical DSN, as shown in Figure 2.1, unembellished sensors are either tethered or in direct contact with a central processor that compiles and interprets the raw detection data. This type of DSN has obvious implications about the range of coverage.

An untethered multi-hop DSN, such as the one in Figure 2.2, has a much

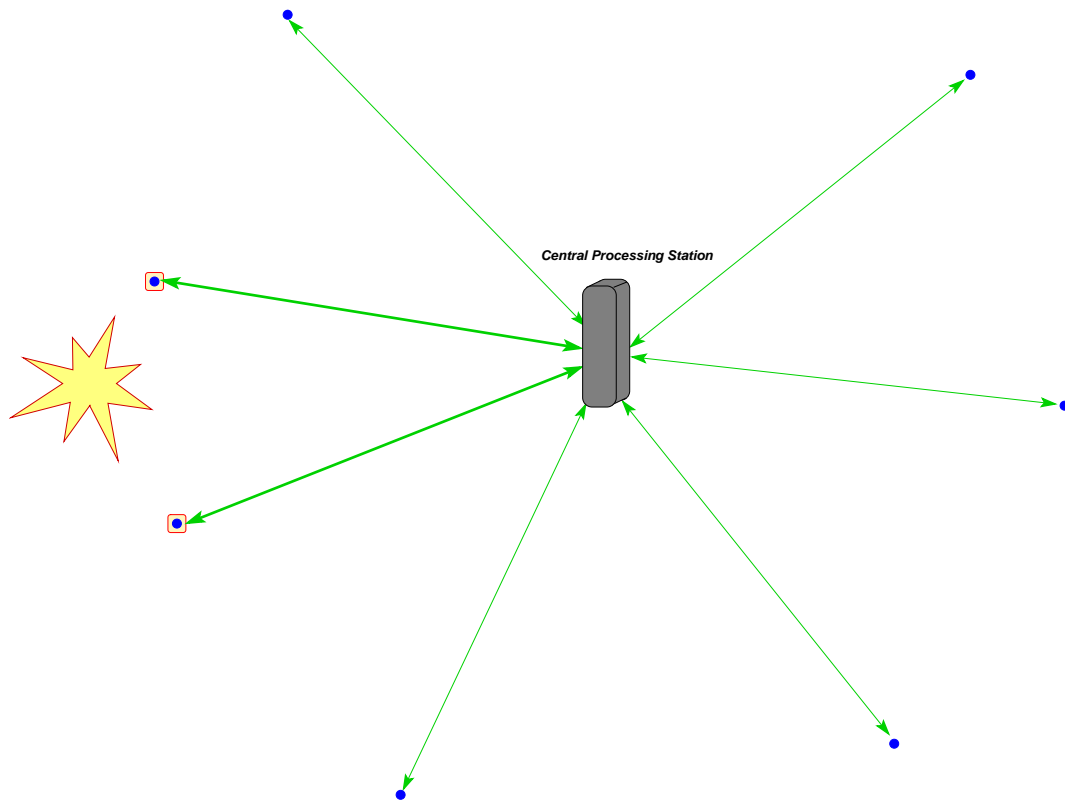


Figure 2.1: Simple Sensors in a Star Configuration

wider range, but is still very dependent on the central processing station and the single point of failure it represents. In addition the sensor nodes are now handling message routing as well as simple detection, and this implies greater computational sophistication, a potential that may be largely untapped.

In situ DSNs are freed from the tyranny of a single processing station. Most or all of the data processing is performed locally by the sensing nodes themselves. Although the data must be retrieved from the network (exfiltrated) somehow, this can occur dynamically from any node by routing the results. In DSNs with Collective Computation (DSN-CC, see Section 2.2.2), each node holds the final global decision once it has been calculated, and so each node can act as an exfiltration point.

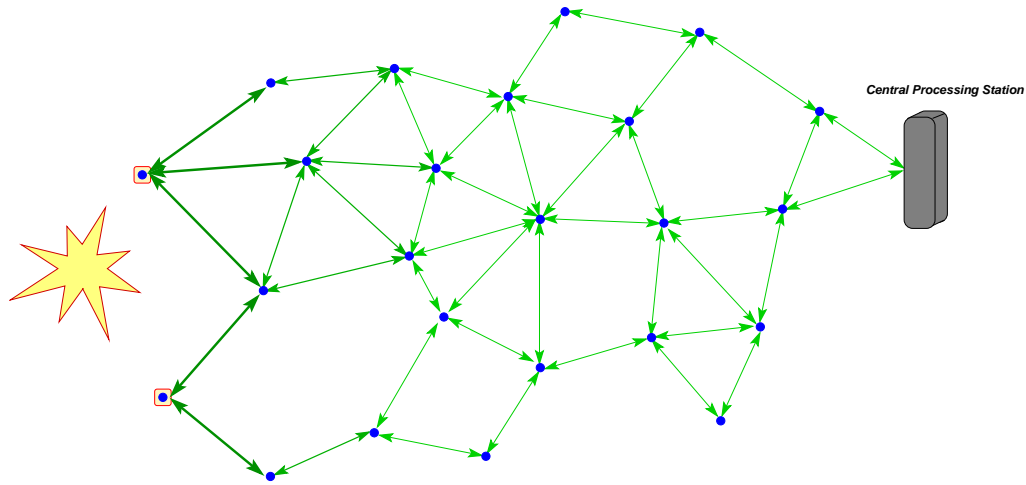


Figure 2.2: Smarter Sensors Slaved to a Compute Center

Node deployment can also affect computation capabilities and not solely by network density. Howard et al. discuss a simple algorithm for mobile sensor blanket coverage over a defined space using potential fields [19]. However, as discussed near the end of Chapter 5, adequate sensor coverage may depend on the phenomena being detected. Typically, however, detection algorithms that are based on unstructured ad hoc node topologies are preferred. Qi et al. point out that optimal sensor placement has been shown to be NP-complete for arbitrary fields [35]. Ad hoc algorithms and methodologies are far more robust and allow for nearly any deployment strategy. Even random scattering is valid so long as RF connectivity constraints are honored.

2.2.2 Computation

Although DSNs could push their raw measurements to some database, as with the Great Duck Island project, in light of the limitations of DSNs such a choice appears to be wasteful. A different processing paradigm is called for in the pursuit of optimal efficiency and by the combination of issues at play in

DSNs: the data volume is much larger, the bandwidth is much lower, there is very limited power, and both the environment and the network are unreliable. Not only is there low bandwidth, but, due to the high power draw of transmission, communication must be reduced in order to prolong node lifetimes. Thus it would be wiser to extract and condense some information from the raw data at the individual nodes.

Data fusion is necessary to some degree in order to maximize network lifespan. Data fusion also implies an added value through multiple sensor readings (achieving quality with quantity). Typically, this would be done in a database approach anyway.

There are a number of approaches to data fusion: fusion of measurements, of data features, or of decisions.

Measurement data fusion combines the raw data. This can be done through coherent sensor data combination, for instance beamforming (a sonar detection technique), or with non-coherent analyses like Bayesian decision-making. Often such algorithms must accommodate scaling issues by first computing in localized clusters, and then expanding outward to a global compute.

Feature level fusion extracts pertinent information from the local data before combining such reduced data sets. A study by González et al. found that similar reductions combined in hierarchical patterns of computation sharing, as opposed to parallel, lower the required bandwidth and processing capacity of a DSN application [11]. This is similar to the local expansion to a global compute suggested above.

To accomplish decision fusion, neighboring nodes share data and cooperatively reach a local decision which is then disseminated throughout the network. A global decision is then reached using these local contributions. This

approach is the focus of the DSN-CC project from Los Alamos National Laboratory (LANL). Note that by passing only conclusions, the total communication requirement is rather low.

One more in situ approach is based on mobile agents. In this strategy, the data is stationary, but the state of the computation moves about the network. Iyengar and Wu [20] hold that this Mobile Agent DSN (MADSN) approach has a low bandwidth requirement, requires little network reliability, is easily extensible, and is very scalable. Mobile agents also have the advantage of *not* discarding any data, which may be crucial in certain applications. Note however that the computation state must be small in order to satisfy the low bandwidth promise, and hierarchical reduction throughout the network will still be necessary to satisfy scaling.

Doubtless in-place computational methods are best for robust, timely results, while maintaining long node lifetimes. DSN-CC and MADSN appear to be complementary approaches and are most likely to apply to the majority of DSN goals.

2.2.3 Data Routing

DSN networking principles, particularly those that are self-organizing, are the most researched sector of DSN properties to date. Popular deployment scenarios tend toward random scattering, and due to power limitations and node vulnerability network reliability is potentially quite low. These ad hoc issues require a communication network capable of repeated self-discovery. Networking research has therefore adopted either light, transitory global structures or is completely structureless. Because both collisions and idle listening waste precious energy, media access control also requires new approaches.

Ad hoc

Ad hoc methods vary from table-driven to on-demand, finding an end-to-end route more or less dynamically. Table-driven ad hoc routing performs a primary, global network discovery to build routing tables for all nodes, then maintains those tables as links are lost or newly discovered. On-demand routing builds and modifies each node's global routing table only as needed: failed routes are modified and retried. Obviously these approaches have opposing assumptions about the frequency of routing between any two end points and the frequency of failure of a route.

As an example, Ad Hoc On demand Distance Vector (AODV) is possibly the most popular ad hoc protocol. AODV maintains a table of routes that is corrected through back-propagation of route error messages. Unknown routes are discovered as needed by flooding to some set radius of nodes. This radius expands, eventually to the entire network, if the route continues not to be found.

Data driven

Data driven approaches include flooding, gossip, rumor, directed diffusion, and min-energy subnet. Flooding and gossip provide examples of how this approach is heedless of the global network since each node knows only of its neighbors.

Flooding very simply rebroadcasts any received message. It is easy to see that this protocol has problems with implosion (duplicate messages), and overlap (the same event is reported by several sensors). These are frequently solved by some form of negotiation, but negotiation also increases overhead.

Haas et al. describe gossip routing as akin to flooding, but the broadcast to

propagate the message only occurs with a probability of between 0.6 and 0.8 in order to improve performance and reduce overall messages [13].

Both ad hoc and data driven approaches also include an alphabet soup of flavors of their various protocols that optimize for shortest path, fault tolerance, low maintenance, energy conservation, and so on.

This wide variety in computation, routing, and positioning strategies begs the question: which is the best? It depends. A MADSN might be concerned with end-to-end routing, while a DSN-CC will not be. Certain DSN applications may be more sensitive to network robustness, energy conservation, etc., requiring different variations in these approaches. Certain DSN infrastructures exist to ease this burden by providing such choices as services.

2.3 DSN Operating Systems

DSN operating system (OS) issues also shed some light on DSN development in general. In particular, the contest between general purpose versus event-driven approaches seems to heavily favor the latter.

The Tiny Microthreading Operating System (TinyOS) is the premier DSN OS and is event-driven. The claim is that this event-driven basis yields a significant improvement in performance, a reduced memory requirement, and reduced power consumption. However it is not necessarily the basis on event interaction that brings about these savings. The fact that TinyOS includes *only* those software modules necessary to complete application tasks is the reason behind its success.

In a comparison of TinyOS against eCOS, an embedded general-purpose OS, Li et al. concluded that the significantly better efficiency of the former was due

to a close match of basic blocks of system behavior with the application's tasks [26]. This is significant because TinyOS is actually not an operating system, but a custom-programming infrastructure.

2.3.1 TinyOS

With the launch of the Smart Dust effort at UC Berkeley in 1999, it rapidly became obvious that the software challenges of hyper-miniaturization would at least equal those of hardware development. To ease the constraints of scant resources as well as to buffer applications from ever-refining hardware platforms, Hill et al. created TinyOS [17].

TinyOS is not a typical OS at all. It is a programming framework for embedded systems and components that eases the creation of a set of application-specific OS-like services.

TinyOS seeks to achieve a small software footprint, low power consumption, efficient concurrency, and high software modularity. Therefore the system is kept simple. The system image consists of a scheduler and a graph of components. Upper level components issue commands to lower layers that in turn issue events to which the upper layers respond. Tasks are posted to the scheduler by commands or events. Executing tasks can only be preempted by an event. Tasks can only operate within the fixed-size memory frame. When the scheduler queue empties and no events occur, the node enters a power-saving state.

Extensions to TinyOS

The TinyOS software distribution also includes some helpful extensions to the basic services.

- TinyDB - a SQL-like query processing interface for data extraction from a TinyOS DSN. It collects data, filters, and aggregates this data in-network, then routes it to an exfiltration PC.
- TinySec - a link layer block cipher using a single symmetric key to solve potential authentication problems in the network. It cannot prevent replay or insider attacks.
- Maté - a virtual machine layered on top of TinyOS [24]. Maté allows viral reprogramming of an entire DSN.

2.3.2 EYES OS

Not to be left out of the DSN arena, the European Union is funding a research project developing collaborative sensor networks (eyes.eu.org). The Energy Efficient Sensor networks (EYES) project is concerned with data services to the end user as well as processing DSN detection data.

The EYES OS architecture consists of three layers: the application, the distributed system services, and the sensor network [6]. The distributed system in turn is composed of a lookup service and an information service. The lookup service handles sensor network configuration and application loading, while the information service collects data together for delivery. Within each individual node is a local information component that allows access to sensory data and a network component that provides protocol stacks.

In practice EYES OS appears to be much like TinyOS in that it is a programming infrastructure for DSNs. This new API approach to operating systems appears to be a distinctive win in providing services for DSN nodes. This approach also suggests that algorithms for use in DSNs should avoid excessive generality as well.

2.4 Fault-Tolerance and Dependability Testing

DSNs may expect nothing about the hospitality of their surroundings, and as a consequence it must be assumed that nodes will fail, and often. Multiple isolated failures, related regional failures, and even rolling blackouts can occur.

Designing for fault-tolerance also affects how DSNs compute results. Fischer et al. long ago demonstrated the need for synchrony to achieve consensus in the face of faults [8]. Due to potentially poor network reliability, it must be assumed that synchrony will be poor as well. Thus fault-tolerant DSN computation cannot require a full consensus.

With an awareness of the failure models that apply in a particular domain, DSN developers can construct fairly thorough tests for use in DSS and even arrive at a confidence bound for a DSN's dependability.

2.4.1 Failure Modes

Failure can occur in the hardware, in the software, in networking, in processing, or in detecting. Regardless of their cause, including intrusion, many of these failures exhibit similar symptoms - ultimately leading to losses in computation or data.

Failures in distributed computing are typically modeled as omission, commission, value, and timing failures. Failure by omission can occur in either the process or the channel. Process omissions are typically fail-stop, which peer nodes can detect, or crashes, which cannot be detected. Send omissions consist of messages that the process has sent but has not placed on the channel. In a receive omission, a node fails to retrieve a message available on the channel.

Errors of commission can likewise occur in the channel or in the process. Send and receive commission faults indicate that data has been sent or received which should not have been. A fault of commission in the process is a spurious occurrence of improper output. A value fault is similar to a process commission fault; it is output that is expected but is incorrect.

Timing faults may involve an individual node's clock or process performance, or the performance of the channel. Timing failures are relevant only in synchronous systems, but as noted in a previous section, most DSNs cannot be synchronous.

Byzantine faults are altogether arbitrary and are generally associated with system intrusion.

Methods of tolerating these faults can be thought of in terms of how they maintain the safety and liveness specifications of an application. Fault-tolerance techniques fall under the major categories of masking, nonmasking, and fail-safe tolerance. In fault masking, both safety and liveness are satisfied even as a fault is occurring. With nonmasking tolerance, safety and liveness are true only after a fault. Fail-safe indicates that only safety is satisfied after the fault.

The remainder of this chapter discusses techniques of masking faults for achieving system dependability.

2.4.2 Dependability Testing

Ideally, a DSN application is not just fault-tolerant but fully dependable. Dependability includes program correctness and security, as well as fault-tolerance. Despite all the techniques to achieve dependability, however, there is only one way to be sure of dependability: test the application. Hamlet proposes that testing can be performed intelligently to “squeeze” failure probability into some low bound [14].

Fault-Tolerance Testing

DSS provides the means of this testing. Through DSS, the DSN developer can purposefully encode any of the failure modes mentioned above. For instance, fail-stop may be simulated by a non-responsive (sleeping) process. Send-omission errors can be simulated with an otherwise correct process that fails to broadcast a message it considers sent. Byzantine errors are the most difficult simulated failures to craft. Their arbitrary nature requires a measure of creativity in order to thoroughly uncover weaknesses in the application.

Security Testing

Security vulnerabilities can also be caught with DSS. A simulated compromised node can launch a variety of attacks on processes and on communication channels. Denial of Service (DoS) threats alone provide a rich source of network disruption. Wood and Stankovic detail a multitude of security threats and DoS possibilities specifically for DSNs [40]. While there is value in securing routing protocols against such threats, an application that can handle regional network DoS failures, such as a black hole where a node advertises a zero cost route from

itself to all other nodes, can also handle the same failure embodied as drained batteries or even node losses from a flood or fire.

Ensuring Correctness

Correctness is, as always, extremely difficult and costly to ensure. Further, code correctness is worthless if there is a fault in the design. While there is a body of literature devoted to the efficacy of correctness testing, decades of lessons learned from software engineering practices indicate that testing is always incomplete. The best that can be done is to lower the bound on the probable presence of faults [14].

Since software faults are inevitable, the only way to be certain that they will not disrupt the system is to catch such errors as they occur. This is accomplished through fault-tolerance techniques which happily handle many of the faults and security issues mentioned above as well as hardware faults.

Software fault-tolerance techniques are a way of masking faults so that the occurrence of an error is not outwardly visible. For real-time systems such as DSNs, this is particularly difficult. DSNs can be classified as real-time because in the general case the large volume of incoming sensor data will not allow DSN applications to linger too long in processing any one data set.

Fault-tolerance techniques must allow the system to maintain availability, reliability, safety, and security. This is achieved through error detection and on-line recovery. Recovery schemes vary widely, but generally they may use single or multiple, possibly diverse, hardware channels, and one or more, possibly diverse, software modules processed either sequentially or concurrently. One method even uses diverse but logically equivalent data sets. Some well-known examples illustrate these principles.

Recovery block (RB) detects errors with an acceptance test. If the acceptance test detects a failure, the system returns to the cached checkpoint and recomputes with an alternate software module that is computationally equivalent yet diverse from the original module. RB has no hardware replication, but both code and time redundancy. This assumes the independence of defects in diverse software, which has been shown to be false; nonetheless the probability of failure is significantly decreased by this method [18]. In addition the acceptance test itself may be faulty.

N-version programming (NVP) uses hardware redundancy to run N diverse modules concurrently. Each module's output should be equivalent given the same input; therefore a majority agreement ought to rule out faulty processes. NVP suffers the same weakness as RB in its core assumption of fault independence [18].

Pradhan and Vidya's Roll-Forward Checkpointing Scheme (RFCS) uses a single software module running concurrently on two processors [33]. If a checkpoint does not match, a spare (third) processor determines which is the valid checkpoint, and the valid processor's *current* checkpoint replaces that of the faulty processor. RFCS cannot detect coding faults without software diversity, but variants exist that use different software modules.

In considering fault recovery, DSNs are a special case in two ways: a) DSN nodes must handle incoming data as well as perform computations and b) although hardware redundancy is available by local neighboring nodes sharing data and processing capability, a 'hot spare' is never a true spare processor since it has its own sensing and processing responsibilities.

Despite these caveats, Xu and Randell present two related real-time recovery schemes that may be adapted to DSNs [41]. Roll-Forward Recovery with Dynamic Replication Checks (RFR-RC) is a modification of RFCS that covers faults

throughout the recovery process with a roll-forward action. RFR-RC avoids rollback, and the associated re-computation, so long as faults do not occur at two consecutive checkpoints (a common assumption), ensuring a timely response. Roll-Forward Recovery with Behavior-Based Checks (RFR-BC) detects faults with a variety of self-detection methods. When a fault has been identified, RFR-BC uses acceptance tests to determine the valid process in the pair. As with RFCS, that valid process's state is transferred to the other processor. RFR-BC avoids the need for a spare processor, but may be unable to detect certain faults and is dependent on a potentially faulty acceptance test.

DSS can easily include RFR-RC and RFR-BC. The key questions that must be studied concern the amount of overhead inherent in these techniques and whether the methods interfere with DSN node responsiveness to incoming data.

Studies of fault-tolerance for DSNs, either in the form of recovery techniques or as some simpler method, do not appear in the literature. There is much work to be done in this field.

As the next chapter demonstrates, DSS specifically enables the developer to explore all these issues to find an optimal DSN for a specific situation.

Chapter 3

Design Approach

With the challenges of DSNs as discussed in the previous chapter, it is clear that current software development tools are inadequate to deal with this new paradigm. Designed primarily for the creation of serial desktop applications, and occasionally for high-performance parallel applications, such tools utterly miss the core live-data-handling issues of DSNs: detection, fusion, routing, and interaction. Unlike common and familiar software products, DSNs are deeply embedded in their surroundings and some may even respond to their environment through robotics.

DSS addresses this challenge. It simulates DSNs *and* the sensory channels they monitor. Because of the embedded nature of DSNs, DSS focuses primarily on the behavior of the network's nodes.

Why simulate? Swain aptly notes: “[A]s the size and complexity of systems increase, simulation is no longer a luxury but a necessity for proper analysis to support good decisions” [37]. Simulations tend to be cheaper, easier, and faster to assemble. Large-scale tests are just as easy and inexpensive as small ones, whereas costs always scale with the size of live tests. Simulations are also

reproducible and controlled, which are critical for analysis. On the down-side, a synthetic environment has the potential to not adequately represent reality, often in unexpected ways. As a result, this caveat must be monitored to discover its degree of pertinence.

In this case, the DSS simulator is meant to be the second step in DSN application development. Once the developer has created and serially debugged a fledgling application from an algorithm, the application can be plugged into DSS in order to explore the effects of variation in, for instance, node topologies or data routing schemes. This process allows the problem instance to be well-understood through testing *before* a developer addresses the idiosyncrasies of hardware implementation.

DSNs are by nature very opaque to scrutiny. This is in part due to their characteristic of being unobtrusive and embedded in the environment, and in part because they are geographically distributed. Therefore visualization of the DSN and its surroundings is important to help establish the context of the DSN for the programmer.

As discussed in the previous chapter, what constitutes a DSN and the range of duties they perform are very diverse. DSS handles this diversity by maintaining high flexibility. This is a guiding principle in the design of DSS, as examined below.

3.1 DSS System Architecture

To keep DSS simulation as general and flexible as possible, the DSS architecture defines node objects, their radios, sensors and actuators, and even the environmental event models at run-time from a collection of configuration files.

In addition to giving the developer full control over every aspect of a simulation, this framework allows for the easy substitution of other, potentially radically different, distributed systems. Further, the modularity of the user interface also allows easy modification of visualization capabilities.

DSS decomposes a generic DSN into basic components: the nodes, a communication channel, and one or more sensory channels. The nodes are further divided into radio (an interface to the communication channel), sensors (interfacing sensory channels), and an algorithm to run.

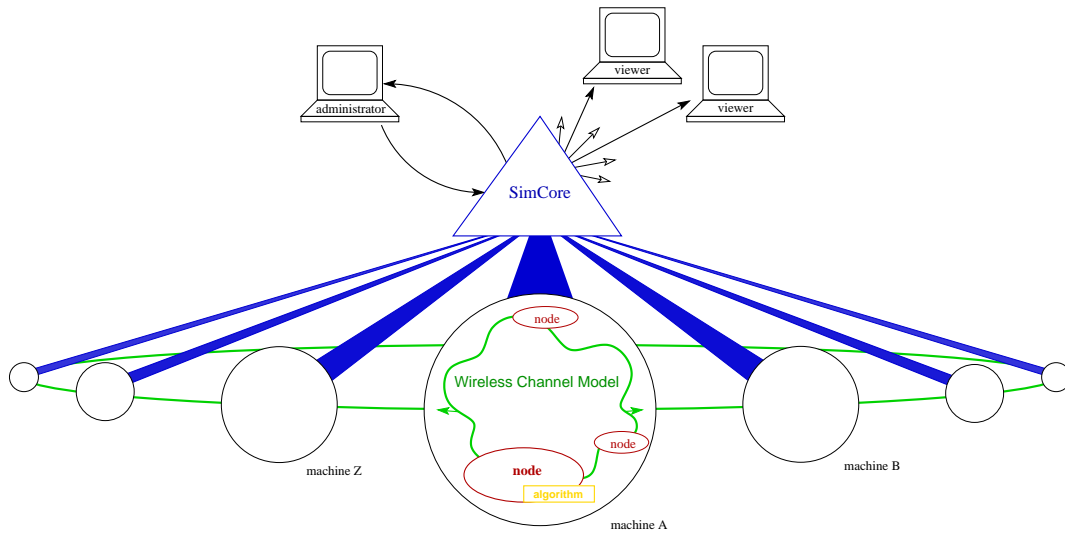


Figure 3.1: DSS Architecture: Overall

3.1.1 User Interface

Figure 3.1 shows only one administrative user. This user configures and starts a DSS simulation. Others may only view the simulation. Additionally, as the figure indicates, the user interface is independent of the simulation, to a large extent decoupled from the intense activity in the core of the simulation.

This decoupling proves to be an important design point for parallelism (see Chapter 4).

3.1.2 SimCore

The blue triangle in Figure 3.1 represents the SimCore, which is the global coordinator for the simulator. This is where the embedded environment is simulated, the configuration stored and shared out, and global timing propagated.

Conceptually the SimCore represents the environment and all the computation that in the real world takes place naturally as the result of physical interactions. Although a physical DSN is interconnected electronically, it is also tied as a whole to its physical surroundings. The architecture of DSS echoes this in that nodes are connected to each other through the Wireless Channel Model (WCM) and are all bound to the SimCore and particularly the environmental simulation (the EnviroSim).

EnviroSim

To produce dynamic detection results across a variety of source types and behaviors, DSS allows the user to specify the propagation behavior of a source and how it is detected by the sensor node. This specification defines the physics that affect phenomena detection calculations.

Many sources are discrete, such as a gunshot, in that a signal is present only for a brief period. These discrete source events may recur, but they are still treated as separate events. Other sources, however, are continuous, such as gamma radiation, and are of long duration. To model this, DSS ‘pushes’ transient detections out to the nodes, but allows the nodes to ‘pull’ in detection data

for continuous sources by querying.

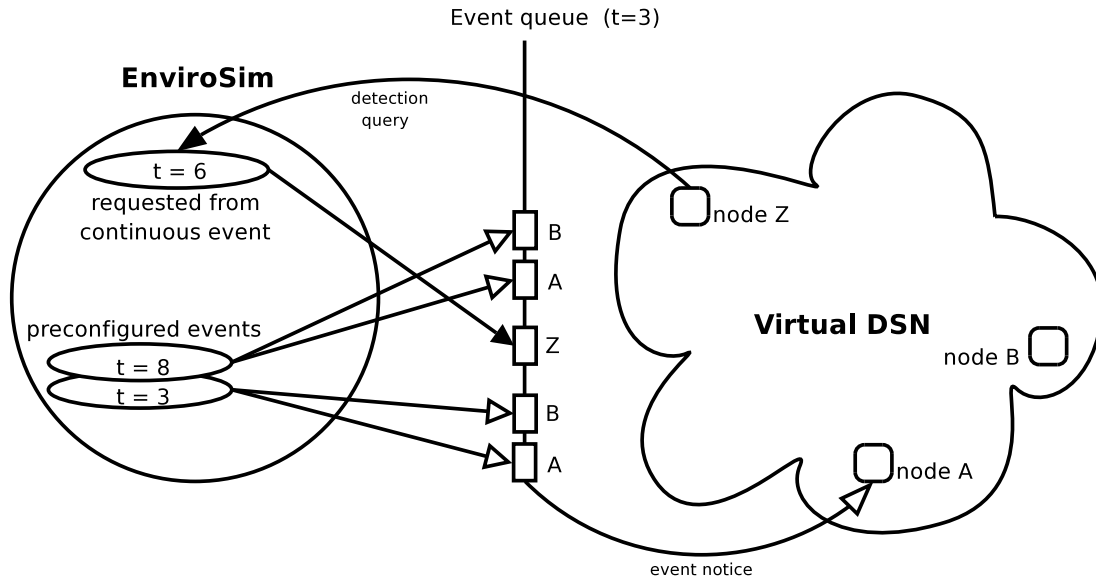


Figure 3.2: EnviroSim Event Queue Interaction

Figure 3.2 demonstrates how DSS orders this pushing and pulling of data for a globally consistent world view of events. Upon initialization, a preconfigured event that occurs at $t = 3$ has its corresponding detection events for nodes A and B calculated and placed on the event queue. The same takes place for the event with $t = 8$. During simulation, the event queue gets to time 3, at which point the detection for node A dequeues and a notice is sent to node A. Meanwhile node Z queries for a detection which is set at $t = 6$; the data for this continuous source is calculated and inserted into the event queue.

When considering node intermessaging as events, it becomes apparent that the ordering of radio messages with source detections could also be important. Timestamping packets solves this potential inconsistency since detection data are already timestamped. The node will be presented with all incoming data in proper global order. How the node deals with that data is up to the application (the ordering may be moot).

Configuration

The internal representation of a DSN in DSS is purposefully abstract. By the grace of this abstraction, the user can specify a wide variety of node topologies, sensor capabilities, radio characteristics, and event sources. All this information is acquired at runtime, dynamically loaded into the simulation and embodied as node processes.

3.1.3 Wireless Channel

To maintain flexibility and most closely echo physical DSNs, DSS only simulates the wireless channel. Further processing, like protocol stacks, must be handled by the nodes themselves - just as with a real DSN. Ideally the WCM should simulate not just radio frequency connectivity and message broadcasting, but also realistic error rates and transmission collisions. Additionally, differences in radio frequency also alter the graph of node connectivity. This graph is passed down from the SimCore, where it is originally calculated. Figure 3.3 enlarges the area in Figure 3.1 that shows the relationship of the WCM to the nodes.

3.1.4 Algorithm Plug-in

The DSS is for naught without a mechanism to allow programmers to insert code in the nodes. Certain interfaces, namely communication and detection, must be owned by DSS for interaction with the simulation, but beyond this application programming interface (API) the developer can create arbitrary functionality. By carefully tying the application's detection handling to source propagation specifications, arbitrary detection behavior can be modeled as well.

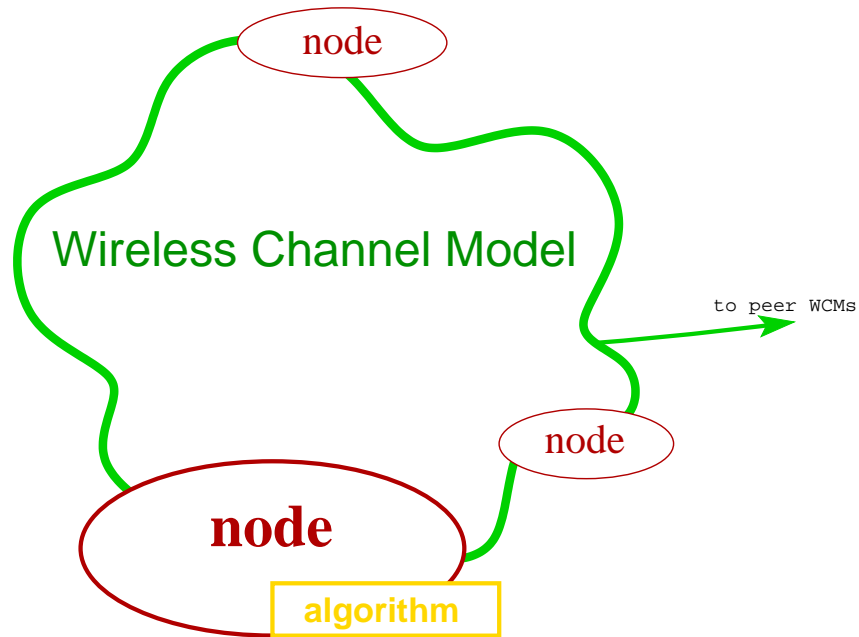


Figure 3.3: DSS Architecture: Remote Machine

This design maximizes the possibility of porting simulation code directly to hardware unchanged.

Distributed Virtually, Distributed Literally

Why the green ring of replicated WCMs in Figure 3.1? With the amount of processing complexity to be serviced, scaling into the thousands of nodes certainly requires special handling. Therefore, DSS allows clustered simulation in parallel. Each circle on this ring represents a subset of DSN nodes interfaced with a WCM. Each WCM communicates only with other peer WCM whose member nodes are directly connected to the local node subset. Cluster distribution, given n WCMs (and processors), n -partitions the DSN connectivity graph such that connections between partitions are minimized. This means that the most highly connected virtual nodes are on a processor together, without im-

balancing the load across all processors. Since typical DSNs are far-flung and sparse, the likelihood of being burdened with a fully connected graph of WCM interactions is low.

By ensuring flexible incorporation of different topologies, sensors, and applications, as well as allowing for a heavy computational load, this design architecture simplifies the implementation decisions covered in the next chapter.

Chapter 4

Implementation

Previous chapters discussed the variety of ways an arbitrary DSN might deal with its situation and data. This chapter will examine how DSS synthesizes all these issues into a single, fully configurable, scalable, and potentially real-time simulation of a DSN embedding.

As Heidemann et al., creators of nam - the ns-2 visualizer, point out [16], the key to accurate simulation is a matter of knowing and/or discovering which details to give greatest credence. Specifically, “[w]hen exploring a new area where many issues are unclear, the need to quickly explore a variety of alternatives can be more important than a detailed result for a specific scenario” [16]. They go on to dub this approach as ‘nimble’ simulation. This principle of agility was the primary consideration in the implementation of DSS, and it allows DSS simulations to embody all the theoretical DSN issues of previous chapters in various combinations.

DSS itself is a careful balance of generalization and specificity, as well as a prime example of agile simulation. Note the extreme configurability of DSS simulations and the allowances for varying detail in the next section.

4.1 Realizing the System Architecture

The design architecture presented in Chapter 3 outlined several main objects. This section will examine some select details of the most important of these. Note from Figure 4.1 how the implementation simply fills in some missing details from the design shown in Figure 3.1.

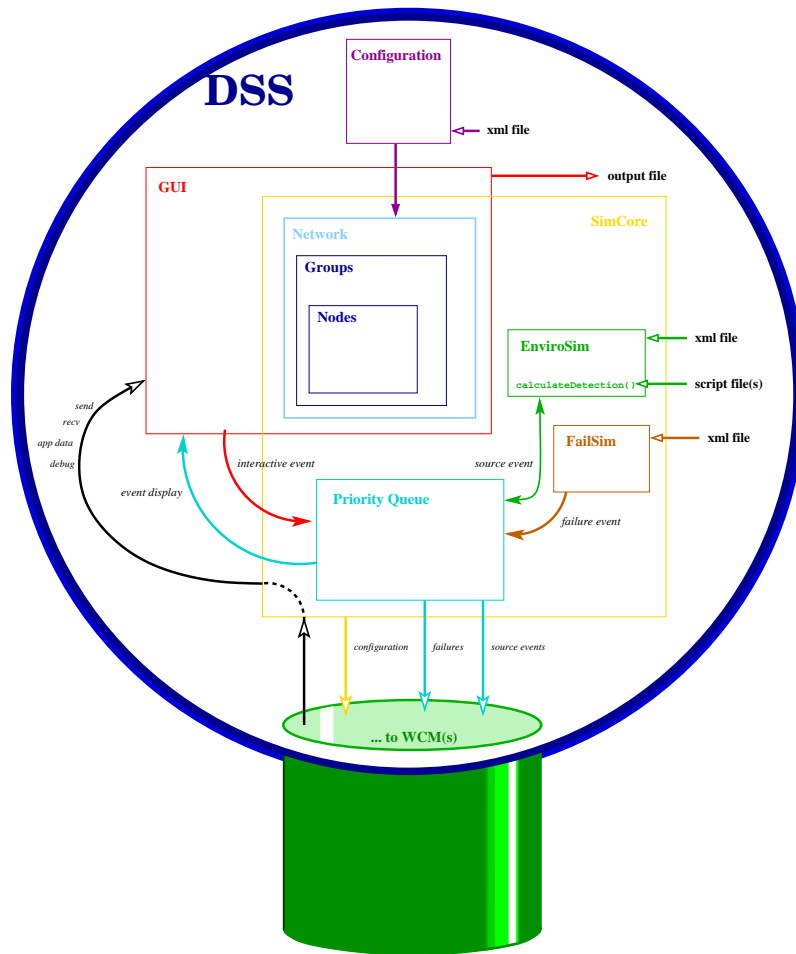


Figure 4.1: Simulator Implementation Objects

All the objects contained in the blue DSS bubble in this figure are implemented in Java. The GUI (Graphical User Interface) object echoes the Network

object and all it contains because the GUI is in fact detached from the SimCore. This is more obvious in Figure 3.1. Figure 4.1 demonstrates that these objects are exactly alike. In fact, in an earlier version of DSS, the Network was a shared object. Although the replication seems like extra work, it is in place to enable remote viewing with a low communication overhead.

The green pipe here leads to one or more WCMs and is established when these C++ executables are spawned. The WCMs in turn are configured, and spawn and configure node executables which run custom applications. All these are interconnected as the design in Chapter 3 dictates.

4.1.1 The User Interface

The interface of DSS is best described in terms of what data the user puts in, sees during a simulation, and eventually gets back out. This overview will introduce many of the components covered in later sections.

Input

Topology This includes the position of each node and specifics such as processing power, radio specification, sensory capabilities, and what application it runs. Nodes that are identical in all but position are configured in groups for the sake of brevity.

Failures Singular or repeated failures are optionally defined, as well as the nature of the failure. Failure simulation consists of a `fork()` and `exec()` over to a selected executable.

Sources Includes the when and where, and the type and strength of an environmental phenomenon.

Phenomena Physics This corresponds with the source types. These scripts dynamically determine just what is detected by a given node and when. The script is given the source's coordinates, the target node's coordinates, the time of the event at the source, and an argument string. Primary in the data string that is returned is the time of detection at that node.

Application The executable to be run in the nodes as defined by the application programming interface (see below).

Display

Node communication The contents of each packet, a brief graphic representation of the transmission range as a sphere, and line indicating connectivity are displayed in various panels.

Source events Sources are indicated as brief points for discrete sources or as connected graphs for moving continuous sources.

Debugging messages As defined by the application developer, and revealed through debugging options, this text appears in its own panel.

Simulation control Start and stop the simulation and display elapsed simulation time.

User-defined messages Any relevant text the application developer chooses can also be displayed.

Output

Simulation record For replay purposes, certain aspects of the simulation are saved to a file. A replay does not invoke node processing or event queuing and thus runs in real time.

User-defined output The application developer may also specify certain information to be saved separately. This is particularly useful for offline analysis.

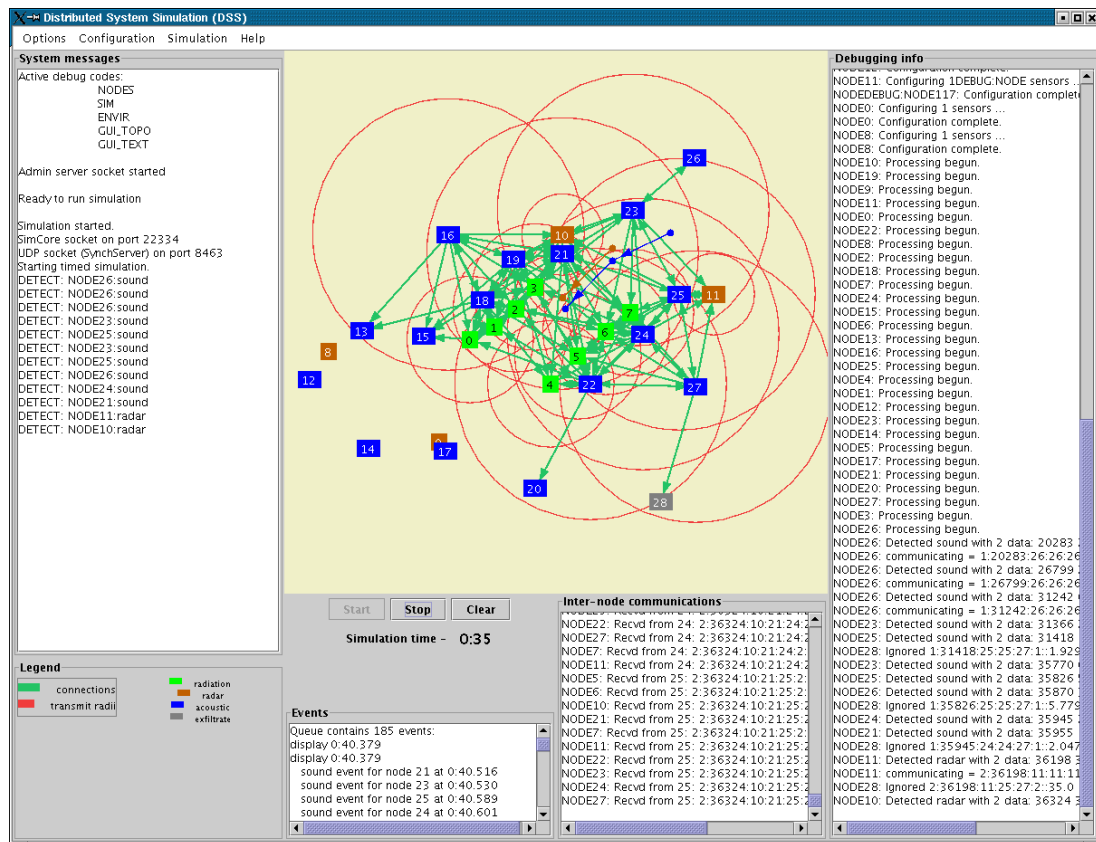


Figure 4.2: DSS Basic Screen

Figure 4.2 shows the basic screen of DSS. Starting clockwise it shows: DSN visualization including node topology, transmissions, connections, and moving sources, the debugging output, the communications packets, the source event queue, and finally system messages and user output.

4.1.2 The Event Priority Queue

Timing is the most crucial detail for DSS. This is not for the sake of real-time viewing, which is most easily achieved by a replay of a simulation. Timing is important because sensors typically care most about *when* an event occurred, and it is vital that the system have coherent global event timing. As a result, the core of the simulator is a Priority Queue Abstract Data Type (ADT). Events are queued with resolution down to the millisecond, as the result of either a discrete source event occurring (such as a single sound) or a node querying for the detection of a continuous source (such as ambient temperature). Discrete events are precomputed for their propagation to each individual node; however detection queries must be handled very rapidly to avoid introducing false latencies. The design architecture requires that DSS have no control over the code that calculates event propagation, although the priority queue should serve out query data as instantly as possible if that is appropriate. The priority queue implementation used in DSS is the Skip List [34], and it is both efficient and relatively simple to code. Being a list implementation, insertion to the front of the list is always only one step, as is delete-min. Thus if event data must go out immediately, it does so; it is added to the queue otherwise - all without any abnormal procedures.

4.1.3 The Scripted EnviroSim

If the event queue is the keystone of DSS, then detection simulation is the cornerstone. Networking simulation has already been done elsewhere and while the simulator gives lip-service to wireless, DSS is really all about distributed detection. This requires simulating the real world in which the nodes exist. Fortunately, these nodes usually use no more than a few sensory chan-

nels. The data reported by a real detector can be as diverse as the vendors that produce the detectors. Therefore, it is the duty of the user to specify just what data is fed to the node as well as the propagation physics of that phenomenon channel. For example, consider the physics associated with an acoustic detector in the air versus one underwater versus one in an air-filled bag underwater. This allows the user to make decisions about details that may or may not impact the performance of a DSN.

This implementation uses scripts written in normal Java code, which are then parsed by the DynamicJava engine by Dyade (koala.ilog.fr/koala/djava). The timing and location of each event is used in calculating the time of detection at each node position within range of a particular phenomenon. This detection data is transferred to the WCM, which then interrupts the target node with a signal and transfers that data onward. Appendix 3 contains a working physics script for basic acoustic detection.

4.1.4 Configuration with XML

Another design point from Chapter 3 involves maximum configurability for node composition, their topologies, event sources, and node failures. The DSS configuration uses the eXtensible Markup Language (XML) to define all four. Why XML? First, implementation was easy. As of Version 1.4, the Java core includes the capability of parsing XML files directly into objects. This gives object persistence that would be otherwise painful and bug-prone to implement. But the use of XML also means that, thanks to its structure, it can be constructed and manipulated in an automated way. This greatly eases the creation of numerous experimental topologies, source configurations, and failure modes for development and research. Further, these XML configurations can be imported from and exported to much more readable forms via eXtensible Stylesheet Lan-

guage (XSL) transformations, which are part of the DSS distribution. This transformation functionality is present thanks to the Xalan module from the Apache Software Foundation (apache.org).

4.1.5 FailSim

Failures are rather simple to simulate. A replacement application is pre-specified in configuration. Fail-stop replaces the working application with one that sleeps until killed. Arbitrary Byzantine failures can be simulated simply by crafting a disastrously similar replacement application. The FailSim module locates the appropriate node process, kills it and spawns the configured replacement which take the place of the original process in the network.

4.1.6 The Wireless Channel Model

The WCM primarily serves as a router for node messages. It is aware of the radio frequency (RF) connectivity, as determined by node location and individual radio ranges, and propagates transmitted packets appropriately. It routes data to nodes on the local host via pipes and ships to remote nodes using sockets to its peer WCMs on other machines. Therefore in cluster simulation, each local WCM has global awareness. The WCM uses `select()` over its node pipes, its peer sockets, and the pipe to the SimCore for the all-important sake of responsiveness to any activity.

4.2 The Application Programming Interface

The nodes are an inherent and inseparable part of the simulator as a whole, yet the DSN developer must be allowed to run any arbitrary program on these virtual nodes. To accomplish this the simulator has a library to which all node executables must link. This library takes care of software objects such as the radio, power source, and a timing link with the simulated environment. The DSN developer simply writes a C++ algorithm object (`Algorithm.cpp`) that either is self-contained or interfaces with existing code. Differing applications, all with the same executable name, are kept separate by the directory hierarchy. The API is rather simple. Communication through the virtual wireless channel takes place by calling `receiv(...)` and `send(...)` in the Radio object. Calling `detectSource(...)` when a blocking radio receive is interrupted, or as otherwise appropriate, retrieves any available sensor data. All this functionality goes in the `nodeProcessing()` method of the Algorithm object. For a more in-depth treatment of the API and further instruction on running DSS, see Appendix 2.

4.3 Features

Although a major design emphasis, network behavior visualization is not critical to simulator performance. In fact, the simulator is perfectly capable of running without the graphical interface. Nonetheless the user can observe networking dynamics, node and source locations, and detection timing graphically. Packet contents, debugging, and other text all scroll by on secondary panels.

There are several debugging options. The most important from the user's point of view is node debugging. However, DSS internal functions can also

be tracked by displaying multiple layers of operational messages for the GUI, SimCore, EnviroSim, Configuration, and WCM objects. Debugging switches are toggled on by command line options.

Large networks and/or high source activity in DSS push processing requirements beyond that which a single processor can handle. Therefore the DSS implementation is distributable over a computational cluster for simulation in parallel. Currently untested, this feature should allow near real-time viewing for complex DSNs on the first run.

Theoretically the simulator is scalable to 63,000 DSN nodes. This has not been tested because such a large network requires cluster computing, which is also untested.

Frequently, cluster head nodes are specialized and not convenient for directly viewing a simulation; hence DSS is remotely viewable. This means that the user interface is separated from the simulator and requires only an IP (internet protocol) address or hostname.

Despite the ease of using XML, creating a complex DSN topology can be slow and error prone. Therefore, DSS also includes interactive DSN configuration, which can then be saved as a correct XML file. Nodes and sources can be positioned and configured graphically.

How does this implementation address the issues of DSNs? Primarily by maintaining flexibility. Any three-dimensional node topology can be loaded in the configuration. Any data processing scheme can be adopted in the application. Any data routing approach can be built on top of the simulated wireless channel. Any node failure or combination of failures can be simulated. Any source phenomena can be detected - assuming its propagation is understood. With some additional work, an operating system such as TinyOS can be in-

cluded.

The modularity of the GUI ensures that the behavioral visualization can be extended or replaced. In fact, the nearly-complete 3D module is a single Java class of reasonable size. With this tremendous flexibility, a single DSN application can be rapidly stress-tested, scale-tested, sensitivity-tested, and efficiency-tested.

This proof-of-concept implementation is naturally immature and does not provide all planned functionality. It in fact assumes a secure, stable, homogeneous network of compute nodes to run on. However, lessons learned from distributed infrastructures such as PVM [9] and CUMULVS [21] indicate the importance of fault-tolerance and security internal to the simulator and are intended for a future release. Chapter 7 describes more future work in store for DSS.

Discovery of which details are relevant to a particular simulation has yet to be addressed. Heidemann et al. reveal that this is done, just as expected, through validation [16]. The next chapter will provide a preliminary validation of DSS.

Chapter 5

Validation Using Experimental Results

Twenty years ago, Robert Sargent, an authority on simulation modeling, presented a tutorial [36] that still serves as a foundation for simulation validation today. Drawing on 30 contemporary papers, his work represents a best-practices consensus on validation methods for simulations. According to Sargent's survey, to verify and validate DSS it is necessary to examine conceptual model validity, verify the computerized model, and establish operational validity. The conceptual model is the design architecture of a simulator. The computerized model involves the programming behind the final executable. These two must match up, which may not always be the case. Finally, operational validity is a matter of ensuring that the output is correct for a given input, and that the operation of the simulator is reasonably equivalent to the real system it simulates. These three areas must be validated conjunctively.

Sargent defines 'Face Validity' as determining "whether the model and/or its behavior is reasonable" [36]. Chapter 3, demonstrates, point for point, how

the design of DSS matched crucial interactions within a general DSN system. This suggests a face validity of the conceptual model of DSS.

The computerized model, by virtue of the use of object-oriented programming, closely follows the design of the conceptual model. As the examples in Chapter 4 make clear, each noun in the design is an object, each verb a method. Hence the validity of the implementation is dependent on the validity of the design. Verification of code correctness is another matter. Sargent contends that extensive testing is the means of verification; yet, the collective experience of developer communities shows that user testing is the most complete means of testing. This requires a user base which DSS does not yet have.

This leaves only operational validity to be demonstrated, which will help reinforce the validity of the conceptual model. Sargent appears to suggest the near self-sufficiency of output behavior testing:

The computerized model is used in operational validity and thus any deficiencies found can be due to an inadequate conceptual model, an improperly programmed or implemented conceptual model on the computer (e.g., due to programming errors or insufficient numerical accuracy), or due to invalid data. [36]

So long as all goes well with an operational comparison of the simulator to a live system, validation is accomplished. The remainder of this chapter will cover the output of a live experimental DSN, the DSS replication of those results, and the differences between the two.

5.1 The Empirical Experiment

Due to the Cartesian nature of DSN data, this section compares experimental (real) data with simulation data graphically. A comparison of these Cartesian points should establish the operational, and hence overall, validity of DSS. This real data comes from an acoustic Time Difference of Arrival (TDOA) experiment at LANL performed in the summer of 2002 by the DSN-CC project [5] .

The LANL TDOA experiment sought to demonstrate the effectiveness of the DSN-CC approach (see Section 2.2.2), test prototype nodes, and benchmark detection performance across a variety of topologies. To this end custom acoustic sensing nodes with processing power and RF communications were built for a live field test. This field test was conducted at the Protection Technology-Los Alamos Live Fire Range, which is used for weapons training for the LANL security force.

The nodes' acoustic sensors were to detect gunshots on the firing range. Node and shooter positions were surveyed from a known exact latitude and longitude about seven kilometers away. The nodes exchange detection data - simply the time at which an acoustic event crosses a particular threshold - until four data points are collected. Then, per the TDOA algorithm (see Appendix 1), from these four measurements a conclusion about the time and location of the event is calculated by hyperbolic triangulation.

The TDOA experiment involved a number of radically different detection topologies used on different days, from which four of the most complete data sets were chosen against which to validate DSS. The results from the day that the DSN-CC team chose to highlight in its report are shown first. The reason why the team chose to highlight this particular data will be obvious upon viewing the results of the remaining three days which follow.

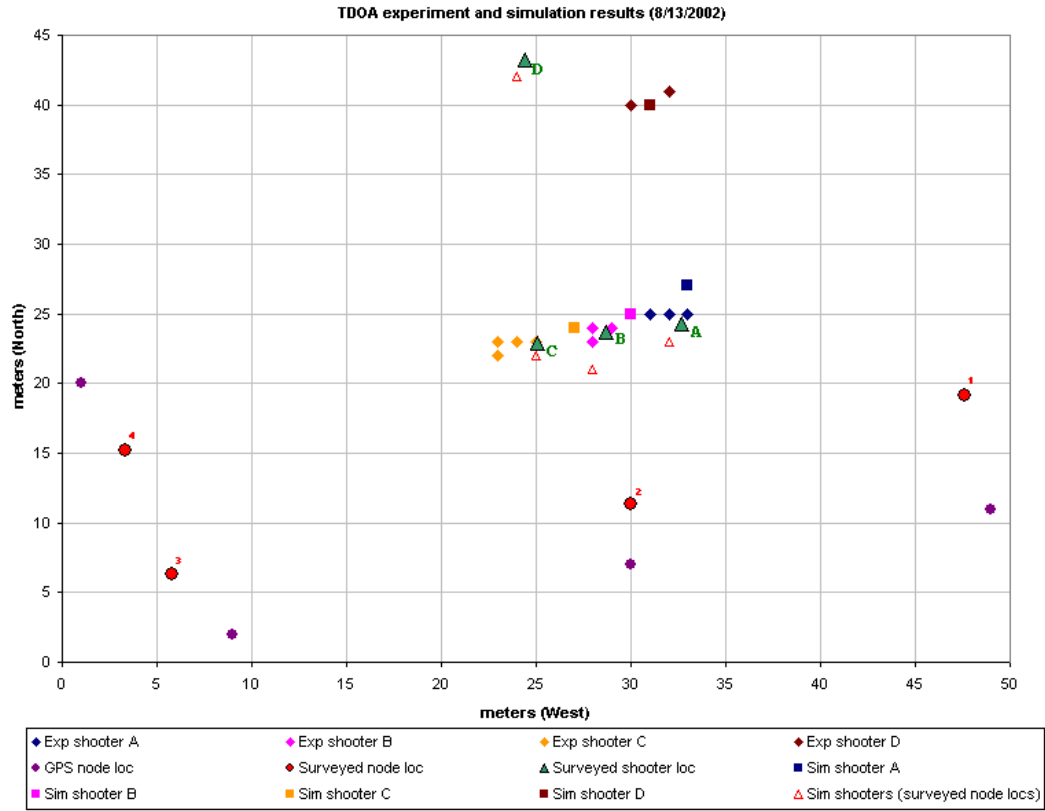


Figure 5.1: TDOA Locating Four Shooters (A, B, C, D)

5.1.1 August 13

The original experiment adjusted the GPS output into the coordinate system of the survey measurements. Thus the coordinates of node 1, for example,

	Source Coordinates							
	A_x	A_y	B_x	B_y	C_x	C_y	D_x	D_y
Surveyed:	32.66	24.24	28.72	23.72	25.07	22.85	24.4	43.2
Calculated:	33	25	28	24	23	22	30	40
	31	25	28	23	24	23	32	41
	32	25	29	24	23	23		
					25	23		

Table 5.1: TDOA Experimental Results: August 13

were $-6944.4 : -855.9$, while its GPS-derived value was $-6991 : -855$. For the purposes of verification of DSS, these values were normalized relative to an origin at $-6992 : -875$, which eases both comprehension of the local coordinate system and entry into the simulator.

There is no scaling in the transference to simulation, so the distances of the original system are preserved. Figure 5.1 plots the various coordinates of interest from Tables 5.1 and 5.2. In examining Figure 5.1, note that the TDOA experiment was subjected to real-world noise in the detection channel; hence there are differing conclusions for different events from the exact same location.

Concerning source D, the DSN-CC team found it "impossible to determine

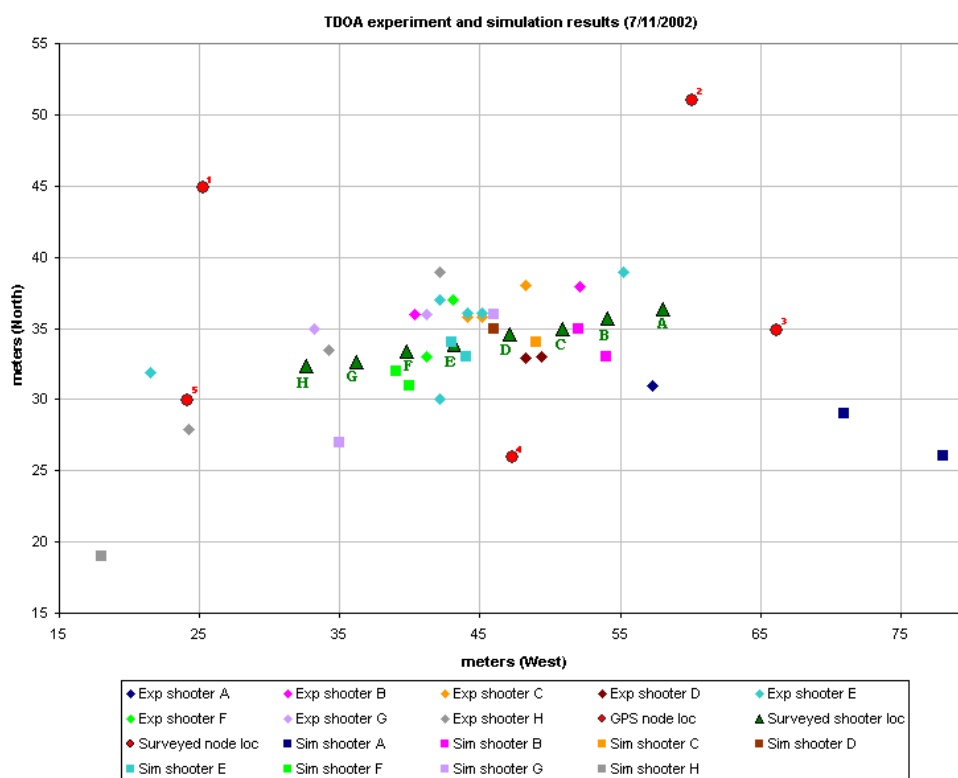


Figure 5.2: TDOA Using Sub-Optimal Topologies: July 11

whether the increased error in location was due to an offset or to increased scatter” [5]. This issue was in such doubt that the team planned for the addition of temperature detection to help establish a more realistic acoustic velocity (it was artificially set at 330 m/s).

As will be demonstrated later, DSS provides some evidence that for this source, this was in fact a very specific offset and that the contribution of any speed-of-sound inaccuracies is insignificant in comparison.

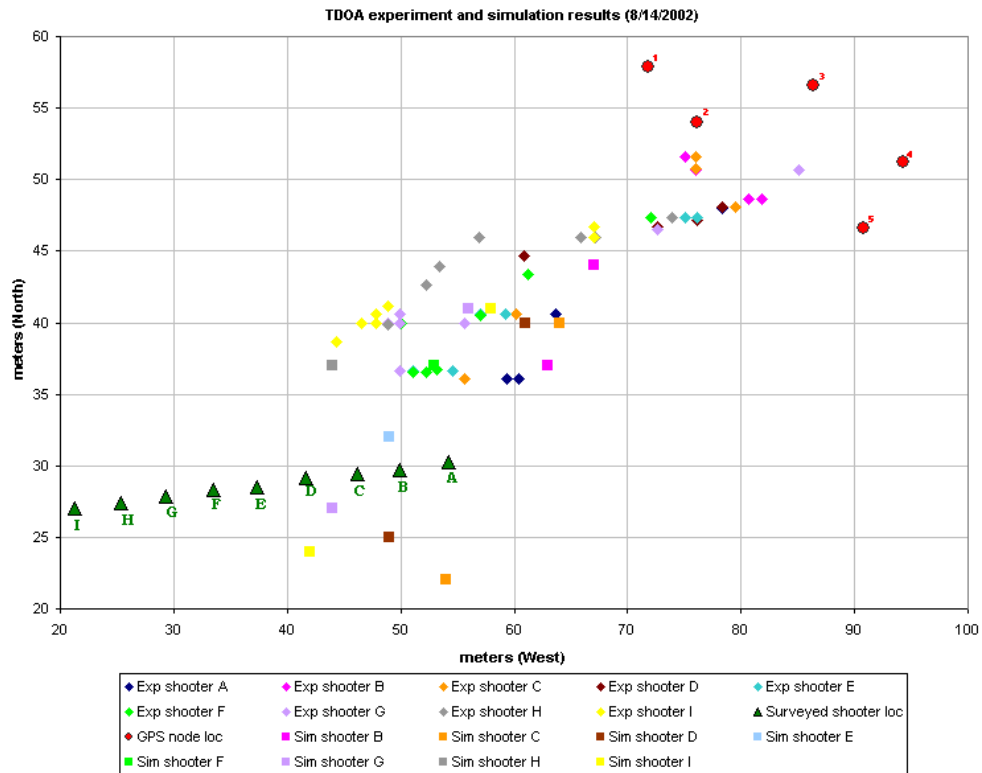


Figure 5.3: TDOA Using Sub-Optimal Topologies: August 14

5.1.2 July 11, August 14, and August 28

Like the August 13 data, these coordinates too were normalized to relatively close origins. Figures 5.2, 5.3, and 5.4 plot the data from these days.

Note that because there are more than four nodes here, it is possible for the network as a whole to produce more than one conclusion for the same acoustic event. Each node collects only three additional data points from the network. Thus where the nodes of the first experiment used all available measurements, these nodes arbitrarily discard one.

Notice that the TDOA algorithm also calculates the time that the event orig-

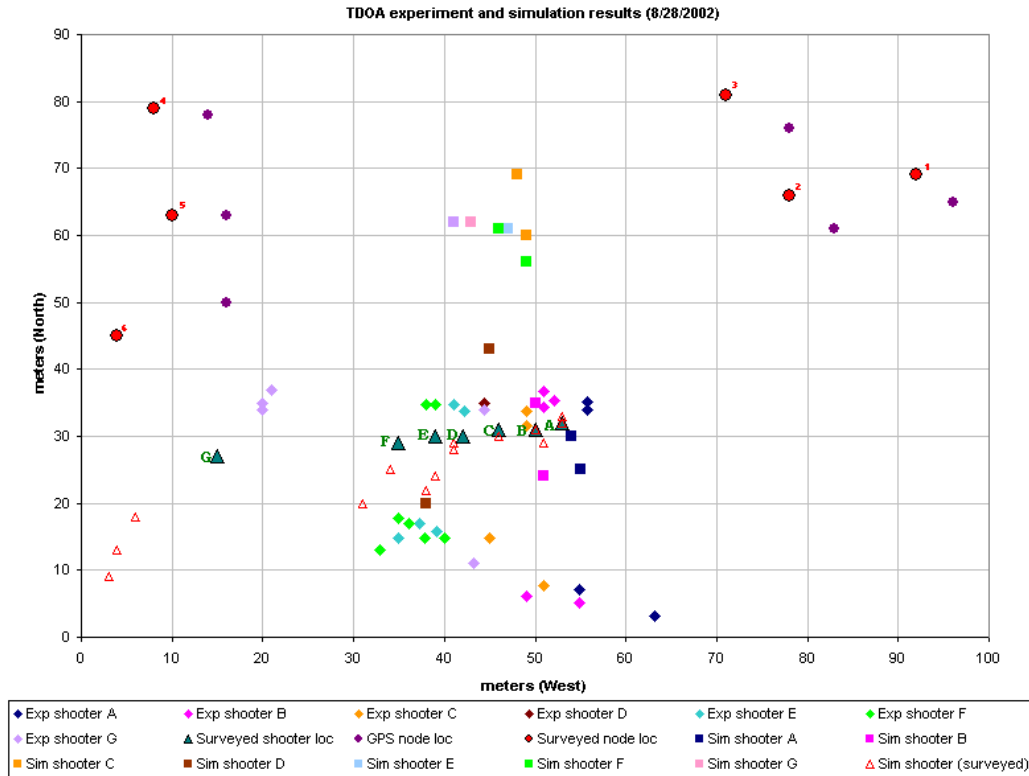


Figure 5.4: TDOA Using Sub-Optimal Topologies: August 28

inated as well as its location. Unfortunately, the TDOA field test was unable to collect actual event timings, and so there is nothing with which to compare. DSS by contrast triggers events with high accuracy at a predetermined time, so a comparison with the original event is possible here. The calculative error is consistently within -50 milliseconds. Without experimental data, it is exceedingly difficult to tell how much of this error is due to the algorithm and how much is due to the simulation. However, if anything, DSS should contribute a positive latency error, not a negative error.

5.2 The Virtual Experiment

The node application for the simulator uses several of the original files used to create the field-experiment node, with minimal changes. The data processing is, aside from hardware differences, exactly the same. The simulation is not subjected to the environmental noise of the original experiment, so every event from the same position yields an identical conclusion data point.

The simulation results which are based on the August 13 data deviated from those of the experiment by at most two meters - except for source C, where the distance of the simulation result from the actual location (2.25 m) is equal to that of the worst experimental distance (2.24 m), but in the opposite direction.

Despite this and the lack of noise noted above, the simulation results are remarkably similar to those of the experiment.

Eliminating the GPS positioning errors by reporting actual positions in the simulator produces some expected improvement, particularly at source D, but source B exhibits a doubling of the initial error (Figure 5.5). Clearly, there are

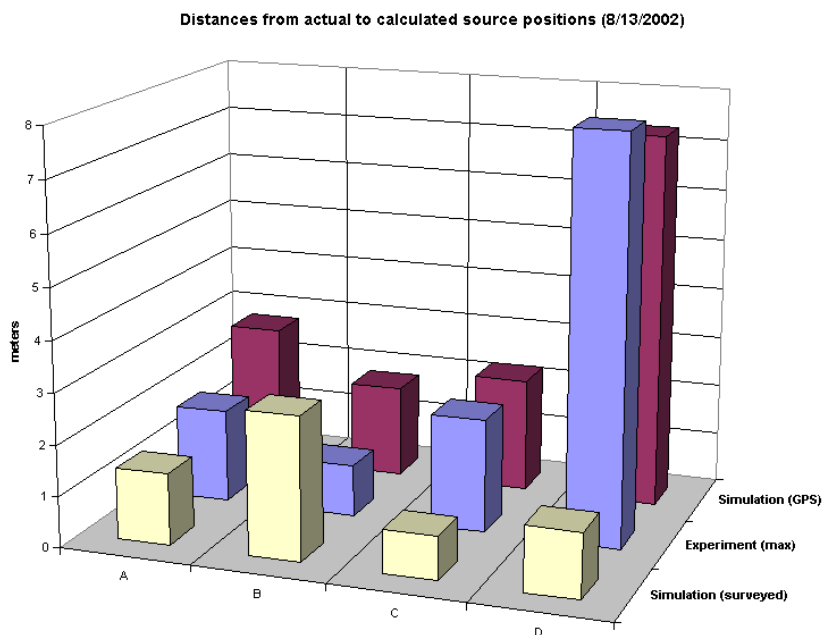


Figure 5.5: Error Distances (August 13)

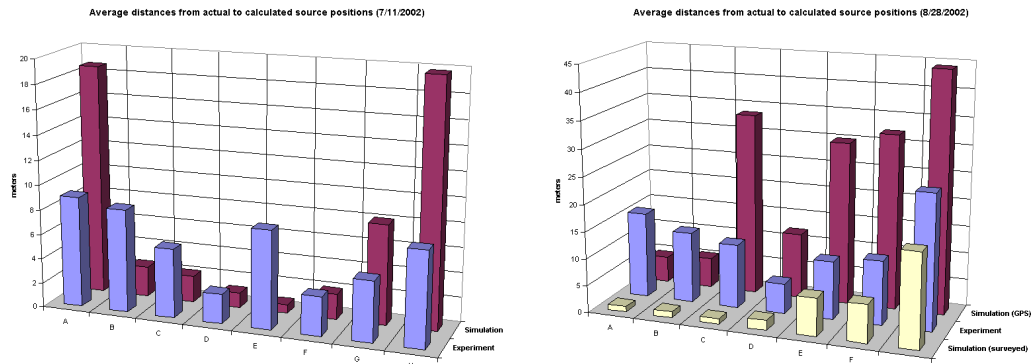
acoustic complexities here, though not resolved in the original work, that are reproduced in the simulator - despite its simplified environmental model.

The graphs of error distances for the data from the remaining days (Figure 5.6) demonstrate a wide variety of behavior. Does this invalidate the simulator? Not after examining the degree of validity of the original data.

The following paragraphs address the acoustic and computational complexities that lead to this divergence among the simulated, the experimental, and the real.

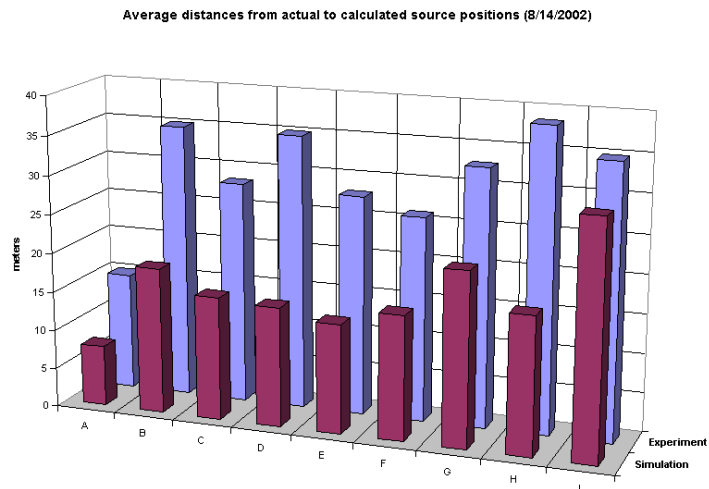
Source Coordinates							
A_x	A_y	B_x	B_y	C_x	C_y	D_x	D_y
31	25	28	23	24	23	32	41

Table 5.2: TDOA Simulation Results: August 13



(a) July 11

(b) August 28



(c) August 14

Figure 5.6: Error Distances

Despite the utterly inconsistent divergence of the simulator's performance from that of the experimental network, there is still some meaning to be found in this data. First, the simulator lacks any noise or echo effects. In the experiment, the complexity that such effects would produce was eliminated by blocking out most of the acoustic detection [5]. This results in nodes capturing measurements that were not necessarily of primary events, whereas the simulator always captured the primary (and only) event. In addition, it turns out that this variety of results is also due to inadequate detector topologies, and that the August 13 topology is the most reasonable. Thus there are otherwise inexplicable results like that of source C on August 28, which were obviously not due to a constant-factor GPS error.

August 13 is a good day with which to compare for two reasons. Firstly, there are only four nodes present and hence all measurements made are used to form a conclusion. With the presence of five or more nodes, measurements are discarded *arbitrarily*. For an optimal topology this is not an issue as any four measurements would give an adequate result, but the simulator simply cannot reproduce the results of a poorly arranged network because it cannot reproduce that (moderately) arbitrary choice. Secondly, the topology is relatively appropriate, and the GPS positioning only reinforces this topology. An optimal topology for this simplified TDOA method would be formed by two symmetric but disjoint linear arrays at right angles. In fact any simple wavefront technique, such as unfiltered TDOA or beamforming, cannot handle significant deviation in detector positioning from exact linearity [4]. Further, a single linear detector array provides bearing but not range. Using two arrays cures this inaccuracy. However, forming an ideal topology on the firing range was a challenge the DSN-CC team was helpless to address since node placement was too severely restricted.

While nonlinear acoustic sensor arrays are possible, and even common in the

literature, use of time difference data requires further error correction, such as Constrained Least Squares (CLS) which then yields the well-known and much more satisfactory TDOA-CLS algorithm [12].

The crucial role of topology is easily shown by running the shooter data from each of the four days through simulation with a six-node topology that follows the x and y axes, with each node spaced 20 meters from its neighbor. The maximum error distance from the actual shooter position in this case was four meters, with an average of *under* two meters for *all* days' data. Additionally, Torney also obtained similar results using the same raw data sets (with the original topologies) passed through a CLS filter [23]. Unfortunately for the DSN-CC team, the TDOA experiment was intended to demonstrate an ad hoc solution; yet the team had an algorithm that did not quite match this goal.

Thus the inability of DSS to produce a perfect reproduction of the results of a real DSN in this case is not a weakness. On the contrary, DSS demonstrates both the significance of the GPS approximation and the lack of robustness in the underlying algorithm - and these are visible even without the experimental data as a comparison. This is precisely what DSS was designed to do and really highlights some of the utility of DSS in uncovering issues of prime importance for both hardware and software development for a specific DSN situation.

Nonetheless, it is obvious that further operational validation using more controlled experiments is needed for DSS.

Chapter 6

Related Work

While there are volumes of work being done on DSNs, including a few simulators, none of it fills quite the same niche as DSS. Most other work tends toward network simulation - which has already been well accomplished by the ns-2 simulator and its wireless extensions. As such, four DSN simulators did serve as inspiration and guidance in designing DSS. This chapter will focus mainly on their divergence from and influence on this project's approach.

6.1 TOSSIM

The TinyOS SIMulator (TOSSIM), earlier known as Nido, simply alters the hardware abstraction layer of TinyOS so that DSN applications can be compiled for and run on a PC. "TOSSIM translates hardware interrupts into discrete simulator events," [25]. TOSSIM, written by Phil Levis of UC Berkeley, is a regular part of the TinyOS distribution.

TOSSIM only allows one application on a virtual DSN of arbitrary scale (thousands of nodes). Like many parallel programs, this application can alter its

behavior by branching as determined by the virtual node's identifier. Networking involves a connected-edge list with bit-error probabilities for each. TinyOS networking stacks handle the rest of the communications details. TOSSIM performance has been tested and found to be "real-time" for "thousands" of nodes; Levis does not quantify this analysis.

TOSSIM does include network visualization through the TinyViz plug-in, but that does not mean that there is any option for node localization data anywhere in this simulator. At least in TinyOS release 1.0, TOSSIM does not inherently include support for any detection simulation. This was in direct conflict with the goals of DSS. Using TOSSIM alone limits the user to exploring only TinyOS. While this is useful, it is too restrictive. TinyOS integration is preferable as a feature, not a design dependency.

6.2 SWAN

The Simulator for Wireless Ad hoc Networks (SWAN), in its TinyOS Scalable Simulation Framework (TOSSF) incarnation, also simulates the operation of TinyOS applications [28], [32]. The principal investigators on this project are L. Perrone and D. Nichol of the Dartmouth College Institute of Security Technology Studies.

SWAN, building on the Dartmouth Scalable Simulation Framework (DSSF), models the RF channel, environmental channels, mobility, terrain, and sensor nodes of a DSN. The terrain governs radio and environmental source propagation and mobile node movement. This model can be two- or three-dimensional, and may potentially allow arbitrary complexity. When [32] was written, the environment model of SWAN consisted of only a gas plume diffusion. Oddly it is "not possible to mix nodes with different mobility models in the same sim-

ulation” [32], which must be a strange implementation. Perrone et al. do not give any details that might explain this. The RF propagation model allows the user to choose among several levels of detail available. The node model is very modular, and similar to DSS it allows easy swapping-in of application code - which is what allowed Perrone et al. to insert TinyOS into their infrastructure relatively well.

SWAN was designed to provide massive scalability to tens of thousands of nodes. There is a great deal of layered complexity in these simulation models in order to provide a vast amount of flexibility, and as such a SWAN simulation of 10,000 sensors over a 100 km² area run for 1000 simulation-seconds requires 10 hours to complete on five processors [28]. SWAN heavily influenced the wireless channel and algorithm plug-in aspects of the DSS design. DSS however seeks to shed some of the complexity of SWAN, as well as increase the ability for distributed simulation computation, therefore achieving more timely results.

6.3 SensorSim

From UCLA, Park et al. composed a framework specifically for DSN simulation [30], [31]. The conceptual model of SensorSim is rather simple: everything is either a node or a channel. “Target” nodes (sources) ‘transmit’ across a sensor channel. This channel governs environmental propagation to sensor nodes which transmit traffic across the wireless channel. The wireless channel likewise controls RF propagation among sensors and to a user node. SensorSim also models hardware and even, rather accurately, battery use within the sensor node. This simulator was used primarily to demonstrate the superiority of time domain multiple access networking coupled with power cycling in conserving power. Such a thorough power model is lacking in DSS; its power

supply handling is rudimentary (see Appendix 2, Section 4.1).

DSS is similar to SensorSim in how it handles point sources. While SWAN is adept with widespread continuous sources, SensorSim appears, in turn, to be limited to discrete sources by design. DSS attempts to deal with both - although plume dispersion is an inherently complex issue. Beyond this, however, SensorSim was not an influence on DSS since the project has been discontinued and the software is no longer available. This project may have been dropped in favor of the recent networking-only GloMoSim produced by the same principal investigators.

6.4 SensorSimII

From Georgia Tech, and unrelated to the UCLA project, Craig Ulmer created an exploration of node clustering techniques, particularly for NASA applications, embodied in SensorSimII [38].

This work provides no obviously new simulation approaches, but it is exemplary of the power of graphical representation. Nodes, links, and transmission radii constantly vary dynamically with network and node activity. This Java applet conceptually guided the graphics implementation of DSS; however the simulation here is tightly bound to the visualization and thus the antithesis to DSS modularity.

Chapter 7

Future Work on DSS

The Distributed Sensors Simulator is barely in beta release, and it is far from mature. The following are a few topics that will be pursued and explored in the coming months, listed in order of highest priority.

7.1 TinyOS Integration

Due to the strong DSN development community's use of TinyOS, the first priority for the next release of DSS is to make it capable of running TinyOS applications in concert with TOSSIM. This will eliminate some of the issues present in TOSSIM as noted in the previous chapter and provide DSS with advanced networking and task scheduling services. Implementing this integration should not be too complicated as TOSSIM provides a socket-based interface for packet injection into its virtual network, which can be modified to use the communication pipes of the WCM inside DSS. With this scheme, many TOSSIM processes will each run a network of one node, and the virtual network will reside in DSS.

7.2 Improved Wireless Channel Realism

Currently DSS uses error-free spherical wireless transmission with a transmission strength fade-off of $1/r^4$ in the simulated radio channel. The nodes are assumed to be on the ground, hence a factor of r^4 as opposed to r^2 . In Heide-mann et al. [16] and elsewhere, it has been established that this approximation is fairly reasonable for simulating radio frequencies outdoors. DSS has no provision for indoor RF characteristics. Further, DSS does not simulate RF transmission collisions in general and specifically neither the hidden nor the exposed station problems.

To address these issues, changes will be made in the WCM module. TOSSIM supposedly simulates hidden stations by modifying node connectivity cells [25]. This is inaccurate since the hidden station problem is only an issue when the interfering node is transmitting. By simply simulating transmission collisions, and the feedback required to detect them, hidden and exposed stations come about naturally. Collision simulation will require a little extra information as well as some closer synchronization inside DSS. Each node will timestamp its transmission and indicate transmission duration. The WCM will monitor incoming transmissions for collisions and will flag packets that have 'collided' as invalid. Nodes will treat these transmissions as garbage. For the timestamping to be effective across a cluster of machines, each machine will have to run the Network Time Protocol (NTP), providing synchronization down to the resolution of DSS: the millisecond. All this adds more complexity to the WCM, but fortunately radios that transmit data on the order of tens of kilobits per second are the norm for DSNs.

For DSS to be helpful in realizing the vision of ubiquity, it must be capable of simulating indoor wireless transmissions. Currently there are two methods

of doing this. Multipath raytracing is very accurate but requires both an accurate model of the environment (see Section 7.4) and intensive computation. Stochastic approximations require much less computation and ignore indoor structures, but also tend to be intolerably inaccurate. As presented by Hassan-Ali and Pahlavan [15], there is a hybrid stochastic method that comprehends the effects of building structure but is not as painstaking as raytracing. Validation by the authors of this method has shown less than 5 dB in error, which is certainly sufficient for DSS. The emphasis of DSS is on behavioral accuracy, not necessarily perfectly simulated networks.

7.3 Robotic Mobility

Static DSNs are certainly very interesting and there is a great deal of work yet to be done. Adding mobility to virtual nodes, however, opens up a whole other realm of possibilities. Mobility and node actuation in general are currently being coded for DSS. Upon completion, DSS will also serve robotics, and particularly micro-robotics, research in much the same ways that it currently serves DSN development.

7.4 Environmental Geometry

Several phenomena and many indoor RF simulations require some knowledge about the geometry of the surrounding environs. Simulated mobile nodes must also be aware of obstacles, even if the applications that direct them are not.

For the two-dimensional case, the solution is as simple as requiring the additional user input of a bitmap of the coordinate field. This is simply the floor plan

of a building or the location of trees and buildings outdoors. This extra input is planned for the next release, primarily to support node mobility. The three-dimensional case will likely require a layered approach, or perhaps something even more complex. It is not expected for the distribution anytime soon.

7.5 Calibration Tools

The need for validation has already been well established. To ease validation against experiments, DSS should also include calibration tools that accept the output of certain live experiments to establish validity in an automated fashion and to make or suggest adjustments where appropriate. Since DSS is simulating a wide variety of hardware, it is useful to make certain adjustments to internal timing elements whenever particular hardware configurations are specified. The true utility of such a tool is yet to be determined.

7.6 Node Processor Emulation

Progressing even further into greater accuracy and greater complexity, one more line of inquiry involves emulating the node processor itself. A similar melding of simulation and emulation for an Atmel 8-bit microcontroller was presented by Adkins and Fait [1]. Enfolding emulation would indeed yield more accurate measurements of response time and power usage and may affect failure recovery and other behaviors. Unfortunately, this may also cut down on scalability and will definitely be limited to those processors for which emulators exist.

There are plenty of other minor changes also on the drawing board. For

instance, random failure is currently determined by a Gaussian distribution. Other random distributions, such as Poisson, will soon be included. However getting too specific in the interests of accuracy may in turn reduce the utility and flexibility of DSS.

Chapter 8

Discussion

This concluding chapter discusses the utility of the DSS GUI in understanding DSN applications, the next step in DSN data handling, and current work with DSS. DSS embodies the unusual approach of *behavioral* software visualization, which helps the developer understand the bigger picture that a given DSN is attempting to capture. What neither DSS nor most DSNs cover is some middle step to get the copious data from the DSN to the user. Although beyond the scope of this work, this issue of information dissemination is worth mentioning because it is so critical. This discussion ends with a description of the latest project with which DSS is assisting.

8.1 Behavioral Simulation and Visualization

The major portion of the software visualization field speaks of animating algorithms and computations, enabling visual programming and development, and navigating through immense data fields [2]. That is not what this project is about; rather the behavioral monitoring described here seeks to encompass

inherent scalability and potential emergence issues, which rarely appear in the visualization literature.

Information visualization involves graphical presentation of very large data sets in an effort to understand coarse-grained trends in the data. Algorithm visualization is frequently used as a teaching tool. These graphic representations frequently reflect the mechanics of a computation as it is performed. Code visualization is essentially more complex forms of pretty printing and cross-referencing. This is not to be confused with visual programming languages that are often based on a schematic approach. Typically these methods attempt to assist in the understanding of complex serial programs.

In contrast, behavioral visualization, at least in the case of DSS, facilitates understanding of simple programs, which are rich in parallel interaction.

Visualization for DSNs is not unlike parallel program visualization. "Understanding parallel programs is more challenging than understanding serial programs because of the issues of concurrency, scale, communications, shared resources and shared state," [22]. These issues are inherent to DSNs as well.

For DSN development, interactive behavior of the system as a whole is the most critical issue in ensuring performance and correctness. Thus DSS seeks to graphically capture significant behaviors in the simulation.

8.2 Information Dissemination

If a DSN is a local area sensing unit, what provides this function for a wide area? Gluing together numerous individual DSNs will not scale well and the fine-grain of the available data will be lost. A structure capable of yielding summary information for a broad geography as well as the details from a single

sensor is called for. Ideally this data collection infrastructure will echo some of the advantages of DSNs: fault-tolerant, decentralized and self-organizing. Currently such distributed data delivery to the end user is fairly under developed.

Bonnet et al. dub data-dumping to a database as a “warehousing” approach and point out the inherent lack of scaling and the energy wastage of such a scheme [3]. These authors propose a device database system that functions much like TinyDB; it enables query processing within the DSN itself. This is also similar to service discovery infrastructures such as Jini from Sun or Universal Plug and Play, which route queries directly to relevant devices. However, these systems are still fairly localized.

Lim presents a distributed services architecture, which is reminiscent of portions of the EYES OS architecture [27]. This three-layered scheme includes a rather detailed application layer and a distributed systems layer composed of a lookup server, a compositional server, and an adaptation server. The lookup server tracks service-provider nodes; the compositional server can reconfigure the services provided by a node or cluster of nodes; the adaptation server handles dynamic topology issues and failure recovery. Although certainly more suited to tying multiple DSNs together, this system also has scaling issues.

These approaches do not suffice to pull together and present information gathered from over a very wide area, such as an entire state or even country. This is a particularly relevant issue in supporting a nuclear detection system as described in the next section.

8.3 Using DSS

DSS is currently being used at LANL to model and explore a nuclear threat reduction scenario.

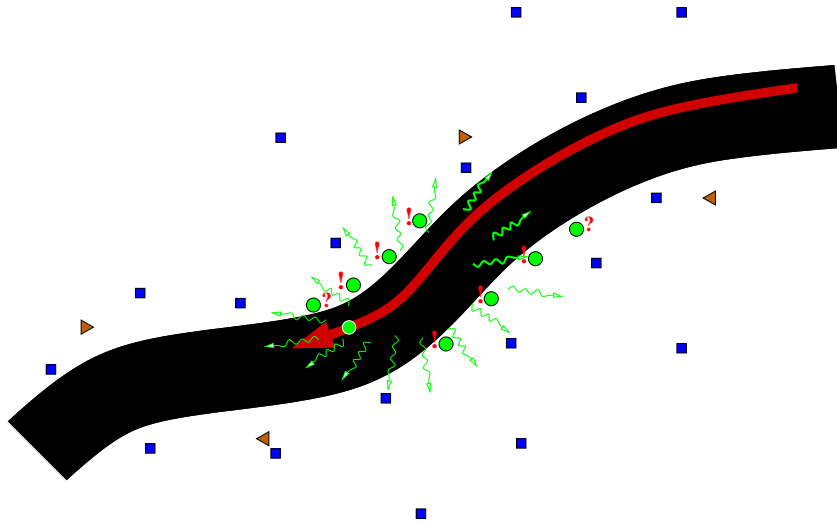


Figure 8.1: Radiation Detection

In this scenario, a certain stretch of highway is being monitored for any movement of radioactive material. This is a heterogeneous DSN, composed of acoustic sensors in blue, radar sensors in brown, and radiation detectors directly beside the road in green.

As a car passes, acoustic and radar sensors determine the size, class, and speed of the vehicle by combining the sensor data intelligently. The radiation detectors (Figure 8.3) pick up gammas from an insufficiently shielded source,

whenever one is present. If this event is determined to be a threat, an alert is routed to an exfiltration point where it is then transmitted to the authorities.

DSS, coupled with nuclear source simulation software, will help investigate sensor size, range, and effectiveness. DSS will also demonstrate the reliability and behavior of data fusion algorithms for detection of special nuclear materials, as well as this scenario's position in the domain of the development issues previously discussed.

Further use and modification of DSS will undoubtedly further strengthen its utility to the DSN community.

The three following appendices consist of a) pseudocode for the basic TDOA algorithm without the CLS filter, b) the informal HOWTO user documentation for DSS in LinuxDoc style, and c) an example Java language script that calculates the time and strength of detection for an acoustic event.

Appendices

Appendix 1

TDOA Algorithm

From Dreicer et al. [5]:

Given a constant signal velocity, the location of a source can be determined by measuring the event propagation time difference at several distributed observation points.

Assuming a constant velocity of, for instance, a sound in air, each pair of observation points (sensor nodes) can constrain the source location (x_0, y_0, t_0) on a hyperbola as

$$[(x_i - x_0)^2 + (y_i - y_0)^2]^{\frac{1}{2}} - [(x_j - x_0)^2 + (y_j - y_0)^2]^{\frac{1}{2}} = v_s(t_i - t_j) \mid i \neq j \quad (\text{A-1.1})$$

The intersection of three hyperbolas which can be determined by four independent measurements can uniquely specify the location of a source. The set of hyperbolas can be summarized as:

$$[(x_1 - x_0)^2 + (y_1 - y_0)^2]^{\frac{1}{2}} - [(x_2 - x_0)^2 + (y_2 - y_0)^2]^{\frac{1}{2}} = v_s(t_1 - t_2) \quad (\text{A-1.2})$$

$$[(x_1 - x_0)^2 + (y_1 - y_0)^2]^{\frac{1}{2}} - [(x_3 - x_0)^2 + (y_3 - y_0)^2]^{\frac{1}{2}} = v_s(t_1 - t_3) \quad (\text{A-1.3})$$

$$[(x_1 - x_0)^2 + (y_1 - y_0)^2]^{\frac{1}{2}} - [(x_4 - x_0)^2 + (y_4 - y_0)^2]^{\frac{1}{2}} = v_s(t_1 - t_4) \quad (\text{A-1.4})$$

Given:

$$A_1 = t_{12}x_{31} - t_{13}x_{21} \quad (\text{A-1.5})$$

$$A_2 = t_{12}x_{41} - t_{14}x_{21} \quad (\text{A-1.6})$$

$$B_1 = t_{12}y_{31} - t_{13}y_{21} \quad (\text{A-1.7})$$

$$B_2 = t_{12}y_{41} - t_{14}y_{21} \quad (\text{A-1.8})$$

$$C_1 = \frac{1}{2}[t_{13}(v_s^2 t_{12}^2 + x_1^2 - x_2^2 + y_1^2 - y_2^2) - t_{12}(v_s^2 t_{13}^2 + x_1^2 - x_3^2 + y_1^2 - y_3^2)] \quad (\text{A-1.9})$$

$$C_2 = \frac{1}{2}[t_{14}(v_s^2 t_{12}^2 + x_1^2 - x_2^2 + y_1^2 - y_2^2) - t_{12}(v_s^2 t_{14}^2 + x_1^2 - x_4^2 + y_1^2 - y_4^2)] \quad (\text{A-1.10})$$

Substituting equations A-1.5 - A-1.10 appropriately:

$$x_0 = (B_2 C_1 - B_1 C_2) / (B_2 A_1 - B_1 A_2) \quad (\text{A-1.11})$$

$$y_0 = (A_2C_1 - A_1C_2)/(A_2B_1 - A_1B_2) \quad (\text{A-1.12})$$

$$t_0 = \frac{1}{4} \sum_{i=1}^4 t_i - [(x_1 - x_0)^2 + (y_1 - y_0)^2]^{\frac{1}{2}} \left(\frac{1}{v_s} \right) \quad (\text{A-1.13})$$

Appendix 2

Distributed Sensors Simulator HOWTO

Distributed sensor networks present a novel and highly complex environment with difficulties and opportunities we are just beginning to explore. The promise of DSNs extends from Homeland Security monitoring to conducting instant and remote inventories to ecological surveys. The Distributed Sensors Simulator (DSS) allows the DSN developer to specify not just the network topology and components, but also customizable node failures and even the very physics of source detection - all in the interests of maintaining maximum flexibility and applicability. This HOWTO will provide a step-by-step guide to running simulations and program debugging within DSS.

1 Introduction

Distributed Sensor Networks are the up-and-coming, number-one means of realizing the vision of ubiquitous computing. Whether you're using TinyOS or creating roll-

your-own DSN services, the Distributed Sensors Simulator can get you up and running sooner and smoother. With DSS, you can develop and debug DSN applications and study the effects of parallel interaction, scaling, and physical embedding - all independently of hardware complications.

For a more in-depth introduction to DSNs see the original Smart Dust site, the TinyOS site, or this updated Smart Dust site.

Programmers can use the infrastructure of DSS to test and debug applications, researchers can conduct algorithm experimentation, and even sensor deployment can be facilitated with DSS since it enables easy visualization of an otherwise opaque operation.

This HOWTO will provide a step-by-step guide to configuring and running DSS.

1.1 Acknowledgments

Thanks go to the organizations that funded this work: namely the NIS-3 and STB-EPO Groups at Los Alamos National Laboratory.

I'd also like to recognize the members of the DSN-CC Project at LANL: Jared Dreicer, Aric Hagberg, Paul Johnson, Angela Mielke, Robert Nemzek, James Rutledge, and David Torney.

Finally and of course most importantly, my graduate advisor, Barney Maccabe, for his invaluable guidance throughout this project.

2 Preparation

Aside from setting up the software, before configuring and programming for DSS, you really should have an embedded situation in mind that you want to simulate. If you are looking to merely familiarize yourself with the project, the distribution includes some sample pre-configured scenarios.

2.1 DSS Setup

So you've downloaded the DSS distribution (from the DSS website), now what?

First you must have installed Java 2 version 1.4 or later - you can get it on the web from java.sun.com.

Next, create a working directory for DSS and unpack the distribution there. Execute **preinstall**, this will add the `$DSSHOME` environment variable to your shell. Run **make**, this will compile the simulator, gui and example applications.

You are now ready to run DSS. All reference to files from here on will be relative to his working directory (i.e. `$DSSHOME`).

2.2 Considerations

Before creating your own DSN, I recommend that you:

Determine Sensor Embedding: This may be as simple as a network of acoustic sensors where the speed of sound is a known constant, or it might be as complicated as imaginable - for instance, neutron detection alone is highly dependent on the geometry of the surrounding landscape which must be modeled for the simulation to have any accuracy. Carefully consider where you will cut corners - and why.

Determine the Network Topology: This is primarily a coverage issue, with an implicit goal of collecting as much data as feasible. Fortunately, the DSN topology is easily changed and can be widely experimented with. Just be aware that topology changes will often also change some operational assumptions.

Determine How to Solve your Problem: Find/create the algorithms and data routing that apply.

Doing all this should ease the steps of configuration.

3 Running DSS

If you are going to configure your DSN by hand, skip to Section 4 for now and return here when you want to run the simulator. If you are a first-time user however, I'd recommend that you read this section now and configure your DSN interactively.

The simulator is disjoint from its user interface for remote viewing and control. You may combine any grouping of XML configurations by entering the name of a configuration file on the command line when invoking the simulator from the working directory (\$DSSHOME), as

```
./DSS-start <config file>
```

The format of this file is simple:

```
XML: dsn-config/xml/<my-topo>.xml
```

```
FAIL: dsn-config/fail/<my-fails>.xml
ENVIR: dsn-config/envir/<my-sources>.xml
OUTPUT: <my-results.out>
```

We will create these XML files in Section 4.

3.1 ... Locally

If you are using just a single machine for both viewing and simulating, you will need two terminals. Again, from the working directory, enter:

```
./DSS-start
```

to start the simulator. Once you see "Initialization successful", in the other terminal type:

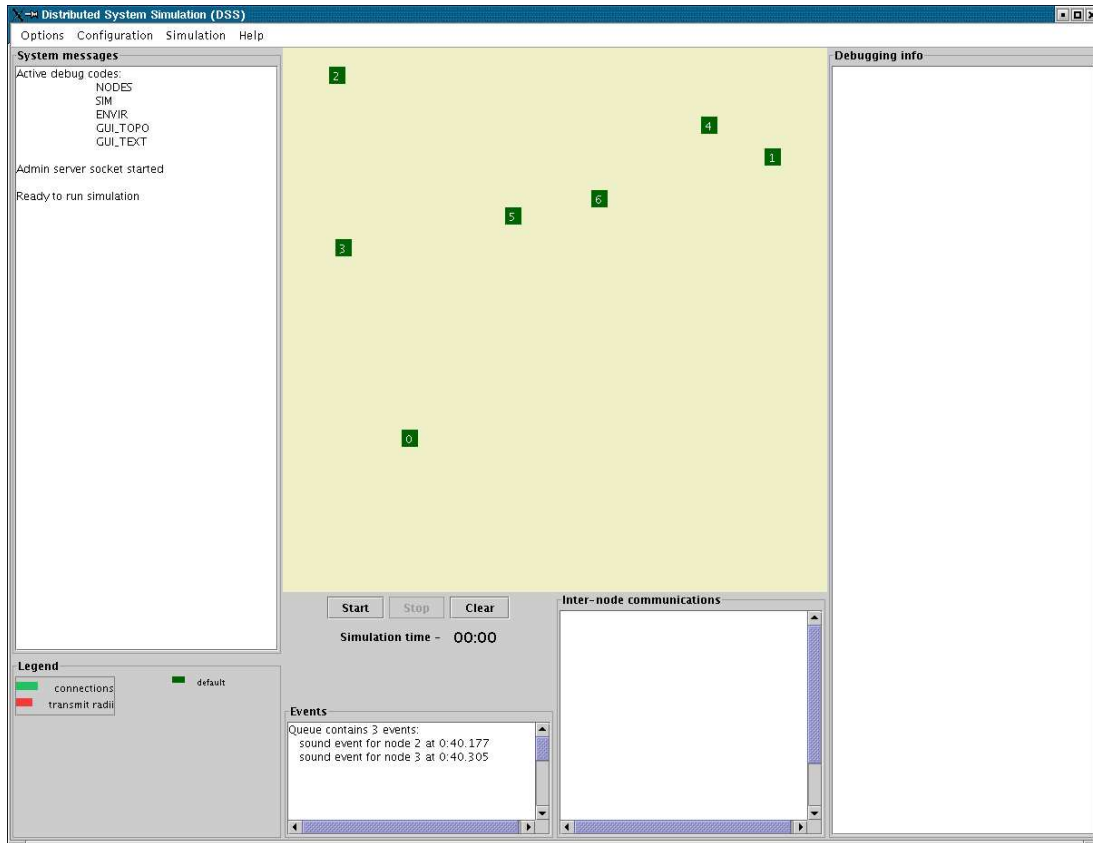
```
./DSS-view
```

and the GUI should start (see Figure A-2.1).

3.2 ... Remotely

Viewing a simulation across a network is almost as easy as it is on a single box. The simulator serves the viewer through ports 25436 and 25438, in case you need to modify a firewall. The simulator is started as above, but the GUI requires that you specify the host that the simulator is on. For example:

Figure A-2.1 Simulation Topology Screen



```
santafe$ ./DSS-start
```

```
magnus$ ./DSS-view santafe
```

Again, waiting for the simulator to issue "Initialization successful" before invoking the viewer.

3.3 Cluster Computing

To simulate over a cluster (and thereby maintain near real-time viewing), the HOSTS must have a listing of each machine to be used. You will need to have SSH setup properly and running ssh-agent for passwordless connections (see the Gentoo Keychain, for an easy solution).

IMPORTANT



Simulation over a cluster is currently unsupported in this release.

4 Configuring the simulation

This is an *ordered* five step process. The simulator requires that certain portions of your DSN configuration be completed before others. Fortunately, the steps get progressively easier.

4.1 Create the Application(s)

Each application you create must have a subdirectory under nodes/, preferably named for the application itself.

Currently the node API supports C++ and (by using "extern C") C code. Your main application code must be in the file `Algorithm.cpp` in your application directory, but may be supported by various other files. The function that handles the main

processing is of course `int nodeProcessing()`. Highlights of the API follow, see `Algorithm.h` for details.

- You must periodically call `mynode->getTimer()->getServerTime()` for synchronization with the sensor embedding.
- Communication takes place using

```
int recvd = mynode->getRadio()->receiv( char* buffer,
                                         int isblocking )
```

and

```
mynode->getRadio()->send( char* buffer, int isblocking )
```

- Detections interrupt blocking radio receives and this data can be retrieved by:

```
if ( recvd < 0 && errno == EINTR )
    int data_points = mynode->detectSource( char* type,
                                           char** data, int seconds );
```

- If you are modeling power usage, radio transmission and reception costs are built-in and depend on the values you enter in the radio configuration (see Section 4.3). Additionally, you must call `mynode->getPower()->use(double mA_drawn)` after every detection, and some number of lines of your code.

To port an existing application, the original `main()` function must be renamed `nodeProcessing()` and put into `Algorithm.cpp`, taking particular care to remove all hardware-specific code.

If you are going to use TinyOS, you must first apply the patch for the release 1.0 distribution (see `nodes/tinyos/patch.readme`). Then copy the `tinyos` directory,

to an appropriately named directory under `nodes/`. Change the `$DSS_ARCH/node` script to point to your TOSSIM application.

IMPORTANT



TinyOS support is not implemented for this beta release

4.2 Compose the Phenomenology Script(s)

These define which nodes detect an event and when. The scripts are written in a Java-like scripting language. The arguments for your scripts will always be (as Java classes):

```
Double Event_X
Double Event_Y
Double Event_Z
Long Time Double Node_X
Double Node_Y
Double Node_Z

String Args
```

The return value is:

String Effect

Since this return value is parsed in the simulator, its structure is moderately strict:

```
"<time of detection(ms)> <data_string_1> ... <data_string_n>"
```

The data strings are packed up into the event that will be shipped to the appropriate node. These are the data points that are read in by `detectSource(...)` (see Section 4.1 for more details).

Beyond these requirements, you may include any arbitrary computation using Java language constructs.

TIP



While you may use the `import` keyword successfully for packages native to the Java distribution, importing external packages usually causes the simulator to fail - this may or may not be a configuration issue to be addressed in a future release.

Here is a simple example script:

```
import javax.vecmath.*;
```

Appendix 2. Distributed Sensors Simulator HOWTO

```
double mps = 343.0;

String Calculate( Double x, Double y, Double z, Long t,
    Double src_x, Double src_y, Double src_z, String args ) {
    double dist = (new Point3d(x,y,z)).distance(new
        Point3d(src_x,src_y,src_z));
    return "" + (t.longValue() + (long)((dist/mps) * 1000));
}

Effect = Calculate( NodeX, NodeY, NodeZ, Time,
    Event_X, Event_Y, Event_Z, Args );
```

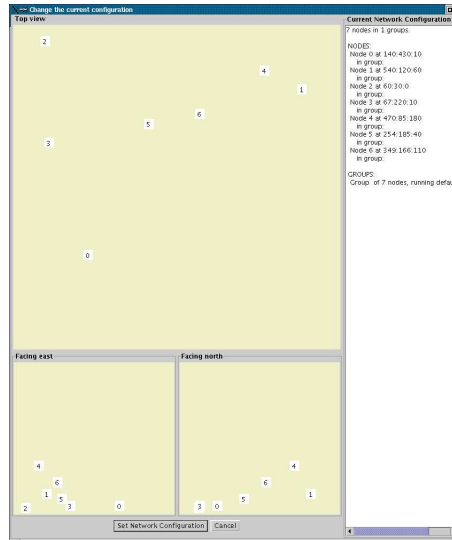
These scripts are located in `dsn-config/phenom/`.

WARNING



The simulation will not tolerate a buggy script. In case of a bug here, watch for Java exceptions somewhere within the `koala.dynamicjava` package.

Figure A-2.2 Network Configuration Screen



4.3 Specify the Node Topology

Interactively

From the menu, select **Configuration** → **New Configuration** to get the network configuration screen (see Figure A-2.2).

Right-click for a menu and **Add a Node**, then fill in the coordinates or randomize.

NOTE



The field is 600 x 600 meters (unless you use the 'zoom' menu option), so scale accordingly.

WARNING



Zooming is not implemented in this release, so be particularly careful when scaling the topology field.

For this first node, there will only be the group 'New'. Clicking **Add Node to Network** will bring up the group configuration (Figure A-2.3). You must name the group and specify the application these nodes will run. You can only choose apps that already exist.

Figure A-2.3 Group Config Screen

Configure a new Node Group

Group name: Application running on all nodes:

RADIO

Id: TR1000

Frequency type: fixed Base frequency: 916.0 MHz

Max range: 30.0 meters Data rate: 40 Kbs

Power draw -

Transmitting: 12.0 mA Receiving: 1.8 mA Sleeping: 1.0E-4 mA

PROCESSOR

Id: ATmega128L Speed: 4 MHz

Memory -

Flash: 128 KB Sram: 4 KB Eeprom: 4 KB

Power draw -

Active: 5.5 mA Sleeping: 0.02 mA

POWER SOURCE

Id: 2xAAA Strength: 3200 mAh Volts: 3.0 volts

You can configure the radio, processor and power source, but the only fields that are currently in use are radio range, power draw for both radio and processor, and the power source strength. Other fields will become effective in future releases, but may now be used as a reference.

Clicking **Add a new Sensor** attaches a sensor as defined in the Phenomenology Script (Section 4.2) to this node. Only those detectors that are predefined as above will

be available from the list.

Actuators are not enabled for this release.

Click **Add Group** and you may now add more nodes to this group or define further groups as above.

XML

Due to the complexity of the network XML file, you can instead import a simpler version. Using the incomplete example below, simply fill in the values for your DSN - expanding with more <nodes> and <groups> as appropriate. To view all possible value options, examine the Network DTD at `dsn-config/util/dsnsim.dtd`. For greater familiarity with XML in general, see XML.com or XML.org.

```
<?xml version="1.0" encoding="UTF-8"?>
<Network name="./dsn-config/xml/default.xml">
  <groups>1</groups>
  <Group>
    <Node id="0" random="false">
      <x>148</x>
      <y>26</y>
      <z>3</z>
    </Node>
    <Node id="1" random="true"/>

    <application>default</application>
    <processor id="CoTS">
      <speed_MHz>100</speed_MHz>
      <flash_KB>64</flash_KB>
```

```
<sram_KB>8</sram_KB>
<power_draw_active_mA>2.1</power_draw_active_mA>
<power_draw_sleeping_mA>0.0070</power_draw_sleeping_mA>
</processor>
<radio id="strong">
  <range_base_meters>150.0</range_base_meters>
</radio>
<sensors>1</sensors>
  <sensor id="0">
    <type>acoustic</type>
  </sensor>
</Group>
</Network>
```

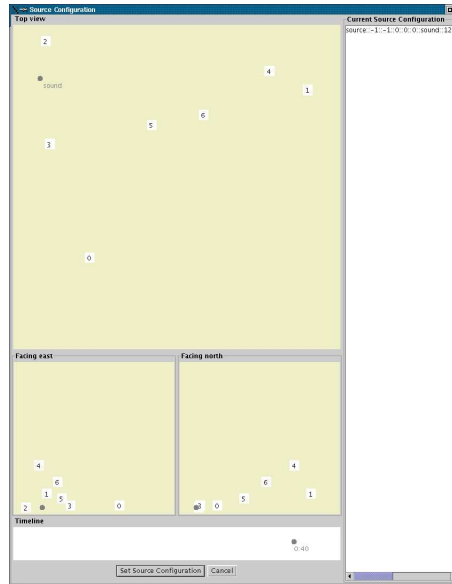
Once complete, import this file by selecting **Configuration** → **Import a Configuration** from the Simulation Topology menu. The simple XML files to import are in `dsn-config/conf/` by default and save as network XML files in `dsn-config/xml/`.

4.4 Specify the Source(s)

Interactively

Right-click anywhere *but* on a node, and select **Configure an Event Source** from either one of the Simulation Topology or the Network Configuration screens. This will spawn the Source Configuration screen (Figure A-2.4).

Figure A-2.4 Source Configuration Screen



Right-click anywhere and select **Add Source**. At this location, you will now have an unconfigured source labeled as such.

Right-click on this source point and select **Configure Source**. The source type must be the name of the corresponding phenomenology script. You may set precise x , y , z and time coordinates here. Any argument string required by the associated physics script. Display color for differentiating between source types. Whether this source is to be continuous. Then **Set Source**.

The source point can be dragged and dropped in the lower panels for y/z , x/z , and time coordinates respectively. In the top panel, dragging and dropping adds a new source moment to a continuous source which can be repositioned via the lower panels.

XML

Because the Source (and the Failure) XML files are much simpler, we don't need the intermediary of importing. Here is a basic single-source XML example:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_01" class="java.beans.XMLDecoder">
  <object class="lanl.dsn.sim.Environment">
    <void property="sources">
      <void method="add">
        <object class="lanl.dsn.sim.Source">
          <void property="args">
            <string>120 35</string>
          </void>
          <void property="seg_times">
            <void method="add">
              <long>40000</long>
            </void>
          </void>
          <void property="seg_x">
            <void method="add">
              <double>50.0</double>
            </void>
          </void>
          <void property="seg_y">
            <void method="add">
              <double>100.0</double>
            </void>
          </void>
          <void property="seg_z">
```

```
<void method="add">
  <double>1.0</double>
</void>
</void>
<void property="start_time">
  <long>40000</long>
</void>
<void property="type">
  <string>sound</string>
</void>
</object>
</void>
</void>
</object>
</java>
```

4.5 Specify any Failures

Interactively

From the Simulation Topology screen, right-click on a node and select **Configure Failure**.

Choosing the Interval button indicates that the failure will repeat - this is particularly useful with randomization as the interval time will change at each fail of the node. Alternatively, the Time button indicates a single failure at this node.

You must define a replacement application - choosing the original will simulate a

restart.

NOTE



You may specify multiple 'failures' to various replacements, thereby simulating recovery - or any other desired application switching behavior.

XML

The XML files that specify failures are in `dsn-config/fail/` and are of the form:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_01" class="java.beans.XMLDecoder">
  <object class="lanl.dsn.sim.FaultList">
    <void property="fails">
      <void method="add">
        <object class="lanl.dsn.sim.Failure">
          <void property="random">
            <boolean>true</boolean>
          </void>
          <void property="time">
            <long>240000</long>
          </void>
        </object>
      </void>
    </object>
  </void>
</java>
```


</java>

You're ready to go - **just click "Start"!**

Appendix 3

Example Phenomenology Script

```
/**
    Template script for defining phenomenal event timing
    arguments are always:
    Double Event_X      |
    Double Event_Y      |   source location
    Double Event_Z      |
    Long Time           |   milliseconds
    Double Node_X       |
    Double Node_Y       |   node location
    Double Node_Z       |
    String Args         |   additional input

    and return variable:
    String Effect
**/

/** sound.djava
    "Errors of the order of 20dBA could be introduced if
    weather is not taken into account."
    i.e. refraction, turbulence, atmospheric absorption
**/

double KELVIN_TEMP( double celcius ) {
    return celcius + 273.16;
}
```

Appendix 3. Example Phenomenology Script

```
    /** Calculating sound velocity */
double VELOCITY( double c ) {
    // physical variables – must be calculated/measured for exactness
    double MOLECULAR_WT_DRY = 0.02895; // kg/mol
    double MOLECULAR_WT_VAPOR = 0.018; // kg/mol
    double MOLECULAR_WT = MOLECULAR_WT_DRY;

    // physical constants
    double ADIABATIC = 1.4005;
    double GAS_CONST = 8.314; // J/mol*K

    return Math.sqrt( ( ADIABATIC * GAS_CONST * KELVIN_TEMP(c) ) / MOLECULAR_WT );
} // m/s

    /** Calculating sound attenuation (over simplified) */
double INTENSITY( double power, double range ) {
    return ( power / ( 4.0 * Math.PI * Math.pow( range, 2.0 ) ) );
} // watts/m^2

double Distance( double x1, double y1, double z1,
                 double x2, double y2, double z2 ) {
    return Math.sqrt( Math.pow((x2 - x1), 2 ) + Math.pow((y2 - y1), 2 ) +
                     Math.pow((z2 - z1), 2 ) );
}

String Calculate( double x, double y, double z, long snd_time,
                 double snd_x, double snd_y, double snd_z, String args ) {
    StringBuffer sb = new StringBuffer();
    double snd_strength = 0.0;
    double celcius_temp = 35.0;
    if ( args != null && args.length() > 0 ) {
        int i=0;
        int j=args.indexOf( " ", 0 );
        if ( j > 0 ) {
            celcius_temp = Double.parseDouble( args.substring( i, j ) );
            i = j+1;
            j=args.indexOf( " ", i );
            if ( j != -1 )
                snd_strength = Double.parseDouble( args.substring( i, j ) );
            else
                snd_strength = Double.parseDouble( args.substring( i, args.length() ) );
        } else
            celcius_temp = Double.parseDouble( args.substring( i, args.length() ) );
    }
```

```
}

double snd_dist = Distance( x, y, z, snd_x, snd_y, snd_z );
if ( snd_dist < 150 ) {
    double snd_mps = VELOCITY( celcius_temp );
    sb.append( "" + ( snd_time + (long)((snd_dist/snd_mps) * 1000 ));
               // time of detection (ms)

    if ( snd_strength != 0.0 )
        sb.append( "_" + INTENSITY( snd_strength, snd_dist ));
               // strength of detection
}
return sb.toString();
}

Effect = Calculate( Node_X.doubleValue(), Node_Y.doubleValue(),
                   Node_Z.doubleValue(), Time.longValue(),
                   Event_X.doubleValue(), Event_Y.doubleValue(),
                   Event_Z.doubleValue(), Args );
```

References

- [1] Dan Adkins and Holly Fait. Power and Time Aware Analysis Tool for TinyOS. Undergraduate project report, 2002.
- [2] Ron Baecker, Chris DiGiano, and Aaron Marcus. Software Visualization for Debugging. *Communications of the ACM*, 40(4), April 1997.
- [3] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the Physical World. *IEEE Personal Communications*, 7:10 – 15, October 2000.
- [4] D. Van Cappel. Target motion analysis using time delays measured from a nonlinear array. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'89)*, volume 4, pages 2724–2727, Glasgow, UK, May 1989.
- [5] Jared S. Dreicer, Sarah S. Giandoni, Paul D. Johnson, Anders M. Jorgensen, Robert Nemzek, Ryan C. Sanchez, and Ralph Stiglich. Distributed sensor network with collective computation report: *In-Situ* acoustic signal triangulation. Technical Report LA-UR-02-6342, Los Alamos National Laboratory, 2002.
- [6] Stefan Dulman and Paul Havinga. Operating System Fundamentals for the EYES Distributed Sensor Network. In *EYES Progress Workshop 2002*, Utrecht, the Netherlands, October 2002.
- [7] Steven D. Feller, Evan Cull, David Kammeyer, and David J. Brady. Argus: A Distributed Sensor Network for Real-time Telepresence. Unpublished.
- [8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), April 1985.

- [9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- [10] Steve Glaser. Advanced MEMS Sensors for Civil Engineering Applications. Proposal for a Small Grant for Exploratory Research.
- [11] V. González, E. Sanchis, G. Torralba, and J. Martos. Comparison of Parallel versus Hierarchical Systems for Data Processing in Distributed Sensor Networks. *IEEE Transactions on Nuclear Science*, 49(2), April 2002.
- [12] Fredrik Gustafsson and Fredrik Gunnarsson. Positioning using time-difference of arrival measurements. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2003)*, Hong Kong, PRC, 2003.
- [13] Zygmunt Haas, Joseph Y. Halpern, and Li Li. Gossip-Based Ad Hoc Routing. In *Proceedings of the IEEE INFOCOM*, New York, NY, June 2002.
- [14] Dick Hamlet. Foundations of Software Testing: Dependability Theory. *Software Engineering Notes (Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering)*, 19(5):128–139, 1994.
- [15] Mudhafar Hassan-Ali and Kaveh Pahlavan. A New Statistical Model for Site-Specific Indoor Radio Propagation Based on Geometric Optics and Geometric Probability. *IEEE Transactions on Wireless Communications*, 1(1), January 2002.
- [16] John Heidemann, Nirupama Bulusu, Jeremy Elson, Chalermek Intanagonwiwat, Kun chan Lan, Ya Xu, Wei Ye, Deborah Estrin, and Ramesh Govindan. Effects of Detail in Wireless Network Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 3–11, Phoenix, Arizona, USA, January 2001. Society for Computer Simulation.
- [17] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S.J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, pages 93–104, 2000.
- [18] Martin Hiller. Software Fault-Tolerance Techniques from a Real-Time Systems Point of View - an Overview. Technical Report 98-16, Chalmers University of Technology, Department of Computer Engineering, November 1998.

- [19] Andrew Howard, Maja J. Matarić, and Gaurav S. Sukhatme. Mobile Sensor Network Deployment Using Potential Fields: a Distributed Scalable Solution to the Area Coverage Problem. In *Proceedings of the Sixth International Symposium on Distributed Autonomous Robotics Systems (DARS02)*, Fukuoka, Japan, June 2002.
- [20] S. Sitharama Iyengar and Qishi Wu. Computational Aspects of Distributed Sensor Networks. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)*, 2002.
- [21] J.A. Kohl and P.M. Papadopoulos. Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS. In *2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.
- [22] Eileen Kraemer and John T. Stasko. Issues in Visualization for the Comprehension of Parallel Programs. In *Third Workshop on Program Comprehension*, pages 116–127, Washington, DC, 1994. IEEE Computer Society Press.
- [23] D.R. Lanman and A.M. Jorgensen. Distributed Sensor Networks with Collective Computation. Technical Report LA-UR-02-3306, Los Alamos National Laboratory.
- [24] Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, December 2002.
- [25] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.
- [26] Suet-Fei Li, Roy Sutton, and Jan Rabaey. Low Power Operating System for Heterogeneous Wireless Communication Systems. In *Workshop on Compilers and Operating Systems for Low Power 2001*, September 2001.
- [27] Alvin Lim. Distributed Services for Information Dissemination in Self-Organizing Sensor Networks. *Journal of the Franklin Institute (Special Issue on Distributed Sensor Networks for Real-Time Systems with Adaptive Reconfiguration)*, 338:707 – 727, September 2001.
- [28] Jason Liu, L. Felipe Perrone, David M. Nicol, Michael Liljenstam, Chip Elliott, and David Pearson. Simulation Modeling of Large-Scale Ad-hoc Sensor Networks. In *Proceedings of the 2001 Simulation Interoperability Workshop*, London, UK, 2001.

- [29] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, September 2002.
- [30] Sung Park, Andreas Savvides, and Mani B. Srivastava. SensorSim: A Simulation Framework for Sensor Networks. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 104–111, Boston, MA, 2000.
- [31] Sung Park, Andreas Savvides, and Mani B. Srivastava. Simulating Networks of Wireless Sensors. In *Proceedings of the 2001 Winter Simulation Conference*, 2001.
- [32] Luiz Felipe Perrone and David M. Nicol. A Scalable Simulator for TinyOS Applications. In *Proceedings of the 2002 Winter Simulation Conference*, 2002.
- [33] Dhiraj K. Pradhan and Nitin H. Vaidya. Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture. *IEEE Transactions on Computers*, 43(10):1163–1174, 1994.
- [34] William Pugh. Skip Lists: a Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668 – 676, June 1990.
- [35] Hairong Qi, S. Sitharama Iyengar, and Krishnendu Chakrabarty. Distributed sensor networks - a review of recent research. *Journal of the Franklin Institute*, 338:655–668, 2001.
- [36] Robert G. Sargent. A Tutorial on Verification and Validation of Simulation Models. In S. Sheppard, U. Pooch, and D. Pegden, editors, *Proceedings of the 1984 Winter Simulation Conference*, Dallas, TX, November 1984.
- [37] James J. Swain. Power Tools for Visualization and Decision-Making. *ORMS Today*, February 2001.
- [38] Craig Ulmer. Wireless Sensor Probe Networks - SensorSimII. <http://users.ece.gatech.edu/~grimace/research/sensorsimii/>.
- [39] Mark Weiser and John Seely Brown. The Coming Age of Calm Technology. Technical report, Xerox PARC, October 1996.
- [40] Anthony D. Wood and John A. Stankovic. Denial of Service in Sensor Networks. *IEEE Computer*, 35(10):54–62, October 2002.

- [41] Jie Xu and Brian Randell. Roll-Forward Recovery in Embedded Real-Time Systems. In *Proceedings of the 1996 International Conference on Parallel and Distributed Systems (ICPADS '96)*, Tokyo, Japan, June 1996.

This report has been reproduced directly from the best available copy. It is available electronically on the Web (<http://www.doe.gov/bridge>).

Copies are available for sale to U.S. Department of Energy employees and contractors from:

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
(865) 576-8401

Copies are available for sale to the public from:

National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22616
(800) 553-6847

